目錄

> This is the Simplified Chinese translation of *mostly-adequate-guide*, thank Professor Franklin Risby for his great work!

关于本书

这本书的主题是函数范式（functional paradigm），我们将使用 JavaScript 这个世界上最流行的函数式编程语言来讲述这一主题。有人可能会觉得选择 JavaScript 并不明智，因为当前的主流观点认为它是一门命令式（imperative）的语言，并不适合用来讲函数式。但我认为，这是学习函数式编程的最好方式，因为：

- 你很有可能在日常工作中使用它

  这让你有机会在实际的编程过程中学以致用，而不是在空闲时间用一门深奥的函数式编程语言做一些玩具性质的项目。

- 你不必从头学起就能开始编写程序

  在纯函数式编程语言中，你必须使用 monad 才能打印变量或者读取 DOM 节点。JavaScript 则简单得多，可以作弊走捷径，因为毕竟我们的目的是学写纯函数式代码。JavaScript 也更容易入门，因为它是一门混合范式的语言，你随时可以在感觉吃力的时候回退到原有的编程习惯上去。

- 这门语言完全有能力书写高级的函数式代码

  只需借助一到两个微型类库，JavaScript 就能模拟 Scala 或 Haskell 这类语言的全部特性。虽然面向对象编程（Object-oriented programing）主导着业界，但很明显这种范式在 JavaScript 里非常笨拙，用起来就像在高速公路上露营或者穿着橡胶套鞋跳踢踏舞一样。我们不得不到处使用 `bind` 以免 `this` 不知不觉地变了，语言里没有类可以用（目前还没有），我们还发明了各种变通方法来应对忘记调用 `new` 关键字后的怪异行为，私有成员只能通过闭包（closure）才能实现，等等。对大多数人来说，函数式编程看起来更加自然。

以上说明，强类型的函数式语言毫无疑问将会成为本书所示范式的最佳试验场。JavaScript 是我们学习这种范式的一种手段，将它应用于什么地方则完全取决于你自己。幸运的是，所有的接口都是数学的，因而也是普适的。最终你会发现你习惯了 swiftz、scalaz、haskell 和 purescript，以及其他各种数学偏向的语言。

Gitbook (更好的阅读体验)

- 在线阅读
- 下载EPUB
- 下载Mobi (Kindle)

目录

第 1 部分

第 2 部分

未来计划

- 第 1 部分是基础知识。这是初版草稿，所以我会及时更正发现的的错误。欢迎

提供帮助！

- 第 2 部分讲述类型类（type class），比如 functor 和 monad，最后会讲到到 traversable。我希望能塞进来一些 monad transformer 相关的知识，再写一个纯函数的应用。
- 第 3 部分将开始游走于编程实践与学院学究之间。我们将学习 comonad、f-algebra、free monad、yoneda 以及其他一些范畴学概念。

第 1 章：我们在做什么？

介绍

你好，我是 Franklin Risby 教授，很高兴认识你。接下来我们将共度一段时光了，因为我要教你一些函数式编程的知识。好了，关于我就介绍到这里，你怎么样？我希望你已经熟悉 JavaScript 语言了，关于面向对象也有一点点的经验了，而且自认为是一个合格的程序员。希望你没有昆虫学博士学位也能找到并杀死一些臭虫（bug）。

我并不假设你之前有任何函数式编程相关的知识——我们都知道假设的后果是什么（译者注：此处原文是“we both know what happens when you assume”，源自一句名言“When you assume you make an ASS of U and ME”，意思是“让两人都难堪”）。但我猜想你在使用可变状态（mutable state）、无限制副作用（unrestricted side effects）和无原则设计（unprincipled design）的过程中已经遇到过一些麻烦。好了，介绍到此为止，我们进入正题。

本章的目的是让你对函数式编程的目的有一个初步认识，对一个程序之所以是函数式程序的原因有一定了解，要不然就会像无头苍蝇一样，不问青红皂白地避免使用对象——这等于是在做无用功。写代码需要遵循一定的原则，就像水流湍急的时候你需要天文罗盘来指引一样。

现在已经有一些通用的编程原则了，各种缩写词带领我们在编程的黑暗隧道里前行：DRY（不要重复自己，don't repeat yourself），高内聚低耦合（loose coupling high cohesion），YAGNI （你不会用到它的，ya ain't gonna need it），最小意外原则（Principle of least surprise），单一责任（single responsibility）等等。

我当然不会啰里八嗦地把这些年我听到的原则都列举出来，你知道重点就行。重点是这些原则同样适用于函数式编程，只不过它们与本书的主题不十分相关。在我们深入主题之前，我想先通过本章给你这样一种感觉，即你在敲键盘的时候内心就能强烈感受到的那种函数式的氛围。

一个简单例子

我们从一个愚蠢的例子开始。下面是一个海鸥程序，鸟群合并则变成了一个更大的鸟群，繁殖则增加了鸟群的数量，增加的数量就是它们繁殖出来的海鸥的数量。注意这个程序并不是面向对象的良好实践，它只是强调当前这种变量赋值方式的一些弊端。

```
var Flock = function(n) {
  this.seagulls = n;
};

Flock.prototype.conjoin = function(other) {
  this.seagulls += other.seagulls;
  return this;
};

Flock.prototype.breed = function(other) {
  this.seagulls = this.seagulls * other.seagulls;
  return this;
};

var flock_a = new Flock(4);
var flock_b = new Flock(2);
var flock_c = new Flock(0);

var result = flock_a.conjoin(flock_c).breed(flock_b).conjoin(flo
ck_a.breed(flock_b)).seagulls;
//=> 32
```

我相信没人会写这样糟糕透顶的程序。代码的内部可变状态非常难以追踪，而且，最终的答案还是错的！正确答案是 `16` ，但是因为 `flock_a` 在运算过程中永久地改变了，所以得出了错误的结果。这是 IT 部门混乱的表现，非常粗暴的计算方式。

如果你看不懂这个程序，没关系，我也看不懂。重点是状态和可变值非常难以追踪，即便是在这么小的一个程序中也不例外。

我们试试另一种更函数式的写法：

```
var conjoin = function(flock_x, flock_y) { return flock_x + floc
k_y };
var breed = function(flock_x, flock_y) { return flock_x * flock_
y };

var flock_a = 4;
var flock_b = 2;
var flock_c = 0;

var result = conjoin(breed(flock_b, conjoin(flock_a, flock_c)),
breed(flock_a, flock_b));
//=>16
```

很好，这次我们得到了正确的答案，而且少写了很多代码。不过函数嵌套有点让人费解...（我们会在第 5 章解决这个问题）。这种写法也更优雅，不过代码肯定是越直白越好，所以如果我们再深入挖掘，看看这段代码究竟做了什么事，我们会发现，它不过是在进行简单的加（`conjoin`）和乘（`breed`）运算而已。

代码中的两个函数除了函数名有些特殊，其他没有任何难以理解的地方。我们把它们重命名一下，看看它们的真面目。

```
var add = function(x, y) { return x + y };
var multiply = function(x, y) { return x * y };

var flock_a = 4;
var flock_b = 2;
var flock_c = 0;

var result = add(multiply(flock_b, add(flock_a, flock_c)), multi
ply(flock_a, flock_b));
//=>16
```

这么一来，你会发现我们不过是在运用古人早已获得的知识：

```
// 结合律（assosiative）
add(add(x, y), z) == add(x, add(y, z));

// 交换律（commutative）
add(x, y) == add(y, x);

// 同一律（identity）
add(x, 0) == x;

// 分配律（distributive）
multiply(x, add(y,z)) == add(multiply(x, y), multiply(x, z));
```

是的，这些经典的数学定律迟早会派上用场。不过如果你一时想不起来也没关系，多数人已经很久没复习过这些数学知识了。我们来看看能否运用这些定律简化这个海鸥小程序。

```
// 原有代码
add(multiply(flock_b, add(flock_a, flock_c)), multiply(flock_a,
flock_b));

// 应用同一律，去掉多余的加法操作（add(flock_a, flock_c) == flock_a）
add(multiply(flock_b, flock_a), multiply(flock_a, flock_b));

// 再应用分配律
multiply(flock_b, add(flock_a, flock_a));
```

漂亮！除了调用的函数，一点多余的代码都不需要写。当然这里我们定义 `add` 和 `multiply` 是为了代码完整性，实际上并不必要——在调用之前它们肯定已经在某个类库里定义好了。

你可能在想“你也太偷换概念了吧，居然举一个这么数学的例子”，或者“真实世界的应用程序比这复杂太多，不能这么简单地推理”。我之所以选择这样一个例子，是因为大多数人都知道加法和乘法，所以很容易就能理解数学可以如何为我们所用。

不要感到绝望，本书后面还会穿插一些范畴学（category theory）、集合论（set theory）以及 lambda 运算的知识，教你写更加复杂的代码，而且一点也不输本章这个海鸥程序的简洁性和准确性。你也不需要成为一个数学家，本书要教给你的编程范式实践起来就像是使用一个普通的框架或者 api 一样。

你也许会惊讶，我们可以像上例那样遵循函数式的范式去书写完整的、日常的应用程序，有着优异性能的程序，简洁且易推理的程序，以及不用每次都重新造轮子的程序。如果你是罪犯，那违法对你来说是好事；但在本书中，我们希望能够承认并遵守数学之法。

我们希望去践行每一部分都能完美接合的理论，希望能以一种通用的、可组合的组件来表示我们的特定问题，然后利用这些组件的特性来解决这些问题。相比命令式（稍后本书将会介绍命令式的精确定义，暂时我们还是先把重点放在函数式上）编程的那种“某某去做某事”的方式，函数式编程将会有更多的约束，不过你会震惊于这种强约束、数学性的“框架”所带来的回报。

我们已经看到函数式的点点星光了，但在真正开始我们的旅程之前，我们要先掌握一些具体的概念。

第 2 章：一等公民的函数

第 2 章: 一等公民的函数

快速概览

当我们说函数是“一等公民”的时候，我们实际上说的是它们和其他对象都一样...所以就是普通公民（坐经济舱的人？）。函数真没什么特殊的，你可以像对待任何其他数据类型一样对待它们——把它们存在数组里，当作参数传递，赋值给变量...等等。

这是 JavaScript 语言的基础概念，不过还是值得提一提的，因为在 Github 上随便一搜就能看到对这个概念的集体无视，或者也可能是无知。我们来看一个杜撰的例子：

```
const hi = name => `Hi ${name}`;
const greeting = name => hi(name);
```

这里 `greeting` 指向的那个把 `hi` 包了一层的包裹函数完全是多余的。为什么？因为 JavaScript 的函数是可调用的，当 `hi` 后面紧跟 `()` 的时候就会运行并返回一个值；如果没有 `()`，`hi` 就简单地返回存到这个变量里的函数。我们来确认一下：

```
hi; // name => `Hi ${name}`
hi("jonas"); // "Hi jonas"
```

`greeting` 只不过是转了个身然后以相同的参数调用了 `hi` 函数而已，因此我们可以这么写：

```
const greeting = hi;
greeting("times"); // "Hi times"
```

换句话说，`hi` 已经是个接受一个参数的函数了，为何要再定义一个额外的包裹函数，而它仅仅是用这个相同的参数调用 `hi`？完全没有道理。这就像在大夏天里穿上你最厚的大衣，只是为了跟热空气过不去，然后吃上个冰棍。真是脱裤子放屁多此一举。

用一个函数把另一个函数包起来，目的仅仅是延迟执行，真的是非常糟糕的编程习惯。（稍后我将告诉你原因，跟可维护性密切相关。）

充分理解这个问题对读懂本书后面的内容至关重要，所以我们再来看几个例子。以下代码都来自 npm 上的模块包：

```
// 太傻了
const getServerStuff = callback => ajaxCall(json => callback(json));

// 这才像样
const getServerStuff = ajaxCall;
```

世界上到处都充斥着这样的垃圾 ajax 代码。以下是上述两种写法等价的原因：

```
// 这行
ajaxCall(json => callback(json));

// 等价于这行
ajaxCall(callback);

// 那么，重构下 getServerStuff
const getServerStuff = callback => ajaxCall(callback);

// ...就等于
const getServerStuff = ajaxCall // <-- 看，没有括号哦
```

各位，以上才是写函数的正确方式。一会儿再告诉你为何我对此如此执着。

```
const BlogController = {
  index(posts) { return Views.index(posts); },
  show(post) { return Views.show(post); },
  create(attrs) { return Db.create(attrs); },
  update(post, attrs) { return Db.update(post, attrs); },
  destroy(post) { return Db.destroy(post); },
};
```

这个可笑的控制器（controller）99% 的代码都是垃圾。我们可以把它重写成这样：

```
const BlogController = {
  index: Views.index,
  show: Views.show,
  create: Db.create,
  update: Db.update,
  destroy: Db.destroy,
};
```

...或者直接全部删掉，因为它的作用仅仅就是把视图（Views）和数据库（Db）打包在一起而已。

为何钟爱一等公民？

好了，现在我们来看看钟爱一等公民的原因是什么。前面 `getServerStuff` 和 `BlogController` 两个例子你也都看到了，虽说添加一些没有实际用处的间接层实现起来很容易，但这样做除了徒增代码量，提高维护和检索代码的成本外，没有任何用处。

另外，如果一个函数被不必要地包裹起来了，而且发生了改动，那么包裹它的那个函数也要做相应的变更。

```
httpGet('/post/2', json => renderPost(json));
```

如果 `httpGet` 要改成可以抛出一个可能出现的 `err` 异常，那我们还要回过头去把“胶水”函数也改了。

```
// 把整个应用里的所有 httpGet 调用都改成这样，可以传递 err 参数。
httpGet('/post/2', (json, err) => renderPost(json, err));
```

写成一等公民函数的形式，要做的改动将会少得多：

```
httpGet('/post/2', renderPost);  // renderPost 将会在 httpGet 中调用，想要多少参数都行
```

除了删除不必要的函数，正确地为参数命名也必不可少。当然命名不是什么大问题，但还是有可能存在一些不当的命名，尤其随着代码量的增长以及需求的变更，这种可能性也会增加。

项目中常见的一种造成混淆的原因是，针对同一个概念使用不同的命名。还有通用代码的问题。比如，下面这两个函数做的事情一模一样，但后一个就显得更加通用，可重用性也更高：

```
// 只针对当前的博客
const validArticles = articles =>
  articles.filter(article => article !== null && article !== undefined),

// 对未来的项目更友好
const compact = xs => xs.filter(x => x !== null && x !== undefined);
```

在命名的时候，我们特别容易把自己限定在特定的数据上（本例中是 `articles` ）。这种现象很常见，也是重复造轮子的一大原因。

有一点我必须得指出，你一定要非常小心 `this` 值，别让它反咬你一口，这一点与面向对象代码类似。如果一个底层函数使用了 `this` ，而且是以一等公民的方式被调用的，那你就等着 JS 这个蹩脚的抽象概念发怒吧。

```
var fs = require('fs');

// 太可怕了
fs.readFile('freaky_friday.txt', Db.save);

// 好一点点
fs.readFile('freaky_friday.txt', Db.save.bind(Db));
```

把 Db 绑定（bind）到它自己身上以后，你就可以随心所欲地调用它的原型链式垃圾代码了。 `this` 就像一块脏尿布，我尽可能地避免使用它，因为在函数式编程中根本用不到它。然而，在使用其他的类库时，你却不得不向这个疯狂的世界低头。

也有人反驳说 `this` 能提高执行速度。如果你是这种对速度吹毛求疵的人，那你还是合上这本书吧。要是没法退货退款，也许你可以去换一本更入门的书来读。

至此，我们才准备好继续后面的章节。

第 3 章: 纯函数的好处

第 3 章：纯函数的好处

再次强调“纯”

首先，我们要厘清纯函数的概念。

> 纯函数是这样一种函数，即相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用。

比如 `slice` 和 `splice`，这两个函数的作用并无二致——但是注意，它们各自的方式却大不同，但不管怎么说作用还是一样的。我们说 `slice` 符合纯函数的定义是因为对相同的输入它保证能返回相同的输出。而 `splice` 却会嚼烂调用它的那个数组，然后再吐出来；这就会产生可观察到的副作用，即这个数组永久地改变了。

```
var xs = [1,2,3,4,5];

// 纯的
xs.slice(0,3);
//=> [1,2,3]

xs.slice(0,3);
//=> [1,2,3]

xs.slice(0,3);
//=> [1,2,3]


// 不纯的
xs.splice(0,3);
//=> [1,2,3]

xs.splice(0,3);
//=> [4,5]

xs.splice(0,3);
//=> []
```

在函数式编程中，我们讨厌这种会改变数据的笨函数。我们追求的是那种可靠的，每次都能返回同样结果的函数，而不是像 `splice` 这样每次调用后都把数据弄得一团糟的函数，这不是我们想要的。

来看看另一个例子。

```
// 不纯的
var minimum = 21;

var checkAge = function(age) {
  return age >= minimum;
};


// 纯的
var checkAge = function(age) {
  var minimum = 21;
  return age >= minimum;
};
```

在不纯的版本中，`checkAge` 的结果将取决于 `minimum` 这个可变变量的值。换句话说，它取决于系统状态（system state）；这一点令人沮丧，因为它引入了外部的环境，从而增加了认知负荷（cognitive load）。

这个例子可能还不是那么明显，但这种依赖状态是影响系统复杂度的罪魁祸首（http://www.curtclifton.net/storage/papers/MoseleyMarks06a.pdf）。输入值之外的因素能够左右 `checkAge` 的返回值，不仅让它变得不纯，而且导致每次我们思考整个软件的时候都痛苦不堪。

另一方面，使用纯函数的形式，函数就能做到自给自足。我们也可以让 `minimum` 成为一个不可变（immutable）对象，这样就能保留纯粹性，因为状态不会有变化。要实现这个效果，必须得创建一个对象，然后调用 `Object.freeze` 方法：

```
var immutableState = Object.freeze({
  minimum: 21
});
```

副作用可能包括...

让我们来仔细研究一下“副作用”以便加深理解。那么，我们在纯函数定义中提到的万分邪恶的副作用到底是什么？“作用”我们可以理解为一切除结果计算之外发生的事情。

“作用”本身并没什么坏处，而且在本书后面的章节你随处可见它的身影。“副作用”的关键部分在于“副”。就像一潭死水中的“水”本身并不是幼虫的培养器，“死”才是生成虫群的原因。同理，副作用中的“副”是滋生 bug 的温床。

> 副作用是在计算结果的过程中，系统状态的一种变化，或者与外部世界进行的可观察的交互。

副作用可能包含，但不限于：

- 更改文件系统
- 往数据库插入记录
- 发送一个 http 请求
- 可变数据
- 打印/log
- 获取用户输入
- DOM 查询
- 访问系统状态

这个列表还可以继续写下去。概括来讲，只要是跟函数外部环境发生的交互就都是副作用——这一点可能会让你怀疑无副作用编程的可行性。函数式编程的哲学就是假定副作用是造成不正当行为的主要原因。

这并不是说，要禁止使用一切副作用，而是说，要让它们在可控的范围内发生。后面讲到 functor 和 monad 的时候我们会学习如何控制它们，目前还是尽量远离这些阴险的函数为好。

副作用让一个函数变得不纯是有道理的：从定义上来说，纯函数必须要能够根据相同的输入返回相同的输出；如果函数需要跟外部事物打交道，那么就无法保证这一点了。

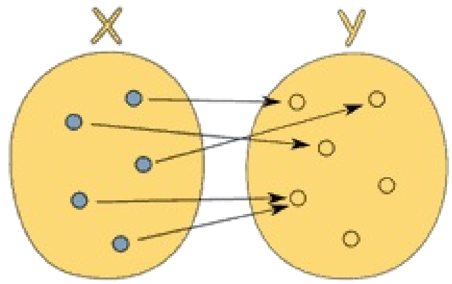
我们来仔细了解下为何要坚持这种「相同输入得到相同输出」原则。注意，我们要复习一些八年级数学知识了。

八年级数学

根据 mathisfun.com：

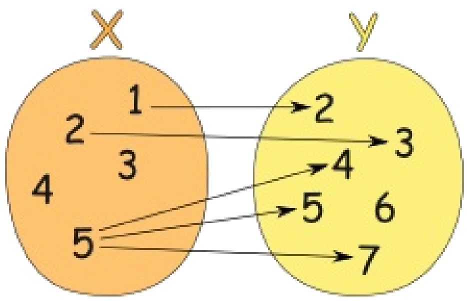
> 函数是不同数值之间的特殊关系：每一个输入值返回且只返回一个输出值。

换句话说，函数只是两种数值之间的关系：输入和输出。尽管每个输入都只会有一个输出，但不同的输入却可以有相同的输出。下图展示了一个合法的从 `x` 到 `y` 的函数关系；



（http://www.mathsisfun.com/sets/function.html）

相反，下面这张图表展示的就不是一种函数关系，因为输入值 `5` 指向了多个输出：



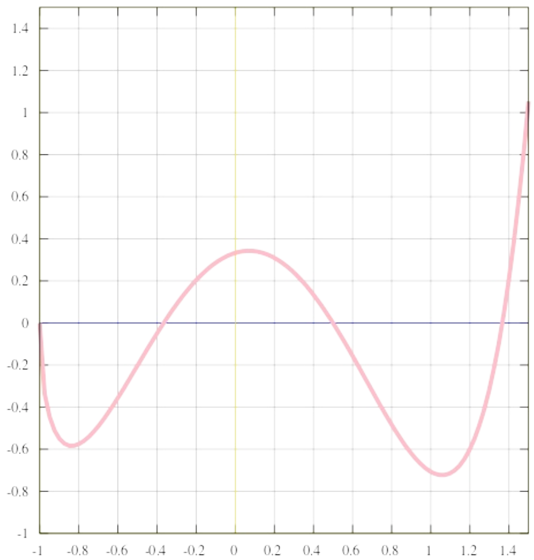
（http://www.mathsisfun.com/sets/function.html）

函数可以描述为一个集合，这个集合里的内容是 (输入, 输出) 对：`[(1,2), (3,6), (5,10)]`（看起来这个函数是把输入值加倍）。

或者一张表：

输入	输出
1	2
2	4
3	6

甚至一个以 `x` 为输入 `y` 为输出的函数曲线图：



如果输入直接指明了输出，那么就没有必要再实现具体的细节了。因为函数仅仅只是输入到输出的映射而已，所以简单地写一个对象就能“运行”它，使用 `[]` 代替 `()` 即可。

```
var toLowerCase = {"A":"a", "B": "b", "C": "c", "D": "d", "E": "e", "D": "d"};

toLowerCase["C"];
//=> "c"

var isPrime = {1:false, 2: true, 3: true, 4: false, 5: true, 6:false};

isPrime[3];
//=> true
```

当然了，实际情况中你可能需要进行一些计算而不是手动指定各项值；不过上例倒是表明了另外一种思考函数的方式。（你可能会想“要是函数有多个参数呢？”。的确，这种情况表明了以数学方式思考问题的一点点不便。暂时我们可以把它们打包放到数组里，或者把 `arguments` 对象看成是输入。等学习 `curry` 的概念之后，你就知道如何直接为函数在数学上的定义建模了。）

戏剧性的是：纯函数就是数学上的函数，而且是函数式编程的全部。使用这些纯函数编程能够带来大量的好处，让我们来看一下为何要不遗余力地保留函数的纯粹性的原因。

追求“纯”的理由

可缓存性（Cacheable）

首先，纯函数总能够根据输入来做缓存。实现缓存的一种典型方式是 memoize 技术：

```
var squareNumber  = memoize(function(x){ return x*x; });

squareNumber(4);
//=> 16

squareNumber(4); // 从缓存中读取输入值为 4 的结果
//=> 16

squareNumber(5);
//=> 25

squareNumber(5); // 从缓存中读取输入值为 5 的结果
//=> 25
```

下面的代码是一个简单的实现，尽管它不太健壮。

```
var memoize = function(f) {
  var cache = {};

  return function() {
    var arg_str = JSON.stringify(arguments);
    cache[arg_str] = cache[arg_str] || f.apply(f, arguments);
    return cache[arg_str];
  };
};
```

值得注意的一点是，可以通过延迟执行的方式把不纯的函数转换为纯函数：

```
var pureHttpCall = memoize(function(url, params){
  return function() { return $.getJSON(url, params); }
});
```

这里有趣的地方在于我们并没有真正发送 http 请求——只是返回了一个函数，当调用它的时候才会发请求。这个函数之所以有资格成为纯函数，是因为它总是会根据相同的输入返回相同的输出：给定了 `url` 和 `params` 之后，它就只会返回同一个发送 http 请求的函数。

我们的 `memoize` 函数工作起来没有任何问题，虽然它缓存的并不是 http 请求所返回的结果，而是生成的函数。

现在来看这种方式意义不大，不过很快我们就会学习一些技巧来发掘它的用处。重点是我们可以缓存任意一个函数，不管它们看起来多么具有破坏性。

可移植性／自文档化（Portable / Self-Documenting）

纯函数是完全自给自足的，它需要的所有东西都能轻易获得。仔细思考思考这一点...这种自给自足的好处是什么呢？首先，纯函数的依赖很明确，因此更易于观察和理解——没有偷偷摸摸的小动作。

```
// 不纯的
var signUp = function(attrs) {
  var user = saveUser(attrs);
  welcomeUser(user);
};

var saveUser = function(attrs) {
    var user = Db.save(attrs);
    ...
};

var welcomeUser = function(user) {
    Email(user, ...);
    ...
};

// 纯的
var signUp = function(Db, Email, attrs) {
  return function() {
    var user = saveUser(Db, attrs);
    welcomeUser(Email, user);
  };
};

var saveUser = function(Db, attrs) {
    ...
};

var welcomeUser = function(Email, user) {
    ...
};
```

这个例子表明，纯函数对于其依赖必须要诚实，这样我们就能知道它的目的。仅从纯函数版本的 `signUp` 的签名就可以看出，它将要用到 `Db` 、 `Email` 和 `attrs` ，这在最小程度上给了我们足够多的信息。

后面我们会学习如何不通过这种仅仅是延迟执行的方式来让一个函数变纯，不过这里的重点应该很清楚，那就是相比不纯的函数，纯函数能够提供多得多的信息；前者天知道它们暗地里都干了些什么。

其次，通过强迫“注入”依赖，或者把它们当作参数传递，我们的应用也更加灵活；因为数据库或者邮件客户端等等都参数化了（别担心，我们有办法让这种方式不那么单调乏味）。如果要使用另一个 `Db`，只需把它传给函数就行了。如果想在一个新应用中使用这个可靠的函数，尽管把新的 `Db` 和 `Email` 传递过去就好了，非常简单。

在 JavaScript 的设定中，可移植性可以意味着把函数序列化（serializing）并通过 socket 发送。也可以意味着代码能够在 web workers 中运行。总之，可移植性是一个非常强大的特性。

命令式编程中“典型”的方法和过程都深深地根植于它们所在的环境中，通过状态、依赖和有效作用（available effects）达成；纯函数与此相反，它与环境无关，只要我们愿意，可以在任何地方运行它。

你上一次把某个类方法拷贝到新的应用中是什么时候？我最喜欢的名言之一是 Erlang 语言的作者 Joe Armstrong 说的这句话：“面向对象语言的问题是，它们永远都要随身携带那些隐式的环境。你只需要一个香蕉，但却得到一个拿着香蕉的大猩猩...以及整个丛林”。

可测试性（Testable）

第三点，纯函数让测试更加容易。我们不需要伪造一个“真实的”支付网关，或者每一次测试之前都要配置、之后都要断言状态（assert the state）。只需简单地给函数一个输入，然后断言输出就好了。

事实上，我们发现函数式编程的社区正在开创一些新的测试工具，能够帮助我们自动生成输入并断言输出。这超出了本书范围，但是我强烈推荐你去试试 *Quickcheck*——一个为函数式环境量身定制的测试工具。

合理性（Reasonable）

很多人相信使用纯函数最大的好处是引用透明性（referential transparency）。如果一段代码可以替换成它执行所得的结果，而且是在不改变整个程序行为的前提下替换的，那么我们就说这段代码是引用透明的。

由于纯函数总是能够根据相同的输入返回相同的输出，所以它们就能够保证总是返回同一个结果，这也就保证了引用透明性。我们来看一个例子。

```
var Immutable = require('immutable');

var decrementHP = function(player) {
  return player.set("hp", player.hp-1);
};

var isSameTeam = function(player1, player2) {
  return player1.team === player2.team;
};

var punch = function(player, target) {
  if(isSameTeam(player, target)) {
    return target;
  } else {
    return decrementHP(target);
  }
};

var jobe = Immutable.Map({name:"Jobe", hp:20, team: "red"});
var michael = Immutable.Map({name:"Michael", hp:20, team: "green"
});

punch(jobe, michael);
//=> Immutable.Map({name:"Michael", hp:19, team: "green"})
```

`decrementHP` 、 `isSameTeam` 和 `punch` 都是纯函数，所以是引用透明的。我们可以使用一种叫做“等式推导”（equational reasoning）的技术来分析代码。所谓“等式推导”就是“一对一”替换，有点像在不考虑程序性执行的怪异行为（quirks of programmatic evaluation）的情况下，手动执行相关代码。我们借助引用透明性来剖析一下这段代码。

首先内联 `isSameTeam` 函数：

```
var punch = function(player, target) {
  if(player.team === target.team) {
    return target;
  } else {
    return decrementHP(target);
  }
};
```

因为是不可变数据，我们可以直接把 `team` 替换为实际值：

```
var punch = function(player, target) {
  if("red" === "green") {
    return target;
  } else {
    return decrementHP(target);
  }
};
```

`if` 语句执行结果为 `false`，所以可以把整个 `if` 语句都删掉：

```
var punch = function(player, target) {
  return decrementHP(target);
};
```

如果再内联 `decrementHP`，我们会发现这种情况下，`punch` 变成了一个让 `hp` 的值减 1 的调用：

```
var punch = function(player, target) {
  return target.set("hp", target.hp-1);
};
```

总之，等式推导带来的分析代码的能力对重构和理解代码非常重要。事实上，我们重构海鸥程序使用的正是这项技术：利用加和乘的特性。对这些技术的使用将会贯穿本书，真的。

并行代码

最后一点，也是决定性的一点：我们可以并行运行任意纯函数。因为纯函数根本不需要访问共享的内存，而且根据其定义，纯函数也不会因副作用而进入竞争态（race condition）。

并行代码在服务端 js 环境以及使用了 web worker 的浏览器那里是非常容易实现的，因为它们使用了线程（thread）。不过出于对非纯函数复杂度的考虑，当前主流观点还是避免使用这种并行。

总结

我们已经了解什么是纯函数了，也看到作为函数式程序员的我们，为何深信纯函数是不同凡响的。从这开始，我们将尽力以纯函数式的方式书写所有的函数。为此我们将需要一些额外的工具来达成目标，同时也尽量把非纯函数从纯函数代码中分离。

如果手头没有一些工具，那么纯函数程序写起来就有点费力。我们不得不玩杂耍似的通过到处传递参数来操作数据，而且还被禁止使用状态，更别说“作用”了。没有人愿意这样自虐。所以让我们来学习一个叫 curry 的新工具。

第 4 章: 柯里化（curry）

第 4 章: 柯里化（curry）

不可或缺的 curry

（译者注：原标题是“Can't live if livin' is without you”，为英国乐队 Badfinger 歌曲 *Without You* 中歌词。）

我父亲以前跟我说过，有些事物在你得到之前是无足轻重的，得到之后就不可或缺了。微波炉是这样，智能手机是这样，互联网也是这样——老人们在没有互联网的时候过得也很充实。对我来说，函数的柯里化（curry）也是这样。

curry 的概念很简单：只传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。

你可以一次性地调用 curry 函数，也可以每次只传一个参数分多次调用。

```
var add = function(x) {
  return function(y) {
    return x + y;
  };
};

var increment = add(1);
var addTen = add(10);

increment(2);
// 3

addTen(2);
// 12
```

这里我们定义了一个 `add` 函数，它接受一个参数并返回一个新的函数。调用 `add` 之后，返回的函数就通过闭包的方式记住了 `add` 的第一个参数。一次性地调用它实在是有点繁琐，好在我们可以使用一个特殊的 `curry` 帮助函数（helper function）使这类函数的定义和调用更加容易。

我们来创建一些 curry 函数享受下（译者注：此处原文是“for our enjoyment”，语出自圣经）。

```
var curry = require('lodash').curry;

var match = curry(function(what, str) {
  return str.match(what);
});

var replace = curry(function(what, replacement, str) {
  return str.replace(what, replacement);
});

var filter = curry(function(f, ary) {
  return ary.filter(f);
});

var map = curry(function(f, ary) {
  return ary.map(f);
});
```

我在上面的代码中遵循的是一种简单，同时也非常重要的模式。即策略性地把要操作的数据（String， Array）放到最后一个参数里。到使用它们的时候你就明白这样做的原因是什么了。

```
match(/\s+/g, "hello world");
// [ ' ' ]

match(/\s+/g)("hello world");
// [ ' ' ]

var hasSpaces = match(/\s+/g);
// function(x) { return x.match(/\s+/g) }

hasSpaces("hello world");
// [ ' ' ]

hasSpaces("spaceless");
// null

filter(hasSpaces, ["tori_spelling", "tori amos"]);
// ["tori amos"]

var findSpaces = filter(hasSpaces);
// function(xs) { return xs.filter(function(x) { return x.match(
/\s+/g) }) }

findSpaces(["tori_spelling", "tori amos"]);
// ["tori amos"]

var noVowels = replace(/[aeiou]/ig);
// function(replacement, x) { return x.replace(/[aeiou]/ig, repl
acement) }

var censored = noVowels("*");
// function(x) { return x.replace(/[aeiou]/ig, "*") }

censored("Chocolate Rain");
// 'Ch*c*l*t* R**n'
```

这里表明的是一种“预加载”函数的能力，通过传递一到两个参数调用函数，就能得到一个记住了这些参数的新函数。

我鼓励你使用 `npm install lodash` 安装 `lodash`，复制上面的代码放到 REPL 里跑一跑。当然你也可以在能够使用 `lodash` 或 `ramda` 的网页中运行它们。

不仅仅是双关语／咖喱

curry 的用处非常广泛，就像在 `hasSpaces`、`findSpaces` 和 `censored` 看到的那样，只需传给函数一些参数，就能得到一个新函数。

用 `map` 简单地把参数是单个元素的函数包裹一下，就能把它转换成参数为数组的函数。

```
var getChildren = function(x) {
  return x.childNodes;
};

var allTheChildren = map(getChildren);
```

只传给函数一部分参数通常也叫做局部调用（partial application），能够大量减少样板文件代码（boilerplate code）。考虑上面的 `allTheChildren` 函数，如果用 lodash 的普通 `map` 来写会是什么样的（注意参数的顺序也变了）：

```
var allTheChildren = function(elements) {
  return _.map(elements, getChildren);
};
```

通常我们不定义直接操作数组的函数，因为只需内联调用 `map(getChildren)` 就能达到目的。这一点同样适用于 `sort`、`filter` 以及其他的高阶函数（higher order function）（高阶函数：参数或返回值为函数的函数）。

当我们谈论纯函数的时候，我们说它们接受一个输入返回一个输出。curry 函数所做的正是这样：每传递一个参数调用函数，就返回一个新函数处理剩余的参数。这就是一个输入对应一个输出啊。

哪怕输出是另一个函数，它也是纯函数。当然 curry 函数也允许一次传递多个参数，但这只是出于减少 `()` 的方便。

总结

curry 函数用起来非常得心应手，每天使用它对我来说简直就是一种享受。它堪称手头必备工具，能够让函数式编程不那么繁琐和沉闷。

通过简单地传递几个参数，就能动态创建实用的新函数；而且还能带来一个额外好处，那就是保留了数学的函数定义，尽管参数不止一个。 下一章我们将学习另一个重要的工具： `组合` （compose）。

第 5 章: 代码组合（compose）

练习

开始练习之前先说明一下，我们将默认使用 ramda 这个库来把函数转为 curry 函数。或者你也可以选择由 lodash 的作者编写和维护的 lodash-fp。这两个库都很好用，选择哪一个就看你自己的喜好了。

你还可以对自己的练习代码做单元测试，或者把代码拷贝到一个 REPL 里运行看看。

这些练习的答案可以在本书仓库中找到。

```
var _ = require('ramda');


// 练习 1
//==============
// 通过局部调用（partial apply）移除所有参数

var words = function(str) {
  return split(' ', str);
};

// 练习 1a
//==============
// 使用 `map` 创建一个新的 `words` 函数，使之能够操作字符串数组

var sentences = undefined;
```

```
// 练习 2
//==============
// 通过局部调用（partial apply）移除所有参数

var filterQs = function(xs) {
  return filter(function(x){ return match(/q/i, x);  }, xs);
};



// 练习 3
//==============
// 使用帮助函数 `_keepHighest` 重构 `max` 使之成为 curry 函数

// 无须改动:
var _keepHighest = function(x,y){ return x >= y ? x : y; };

// 重构这段代码:
var max = function(xs) {
  return reduce(function(acc, x){
    return _keepHighest(acc, x);
  }, -Infinity, xs);
};



// 彩蛋 1:
// ============
// 包裹数组的 `slice` 函数使之成为 curry 函数
// //[1,2,3].slice(0, 2)
var slice = undefined;



// 彩蛋 2:
// ============
// 借助 `slice` 定义一个 `take` curry 函数，该函数调用后可以取出字符串的
前 n 个字符。
var take = undefined;
```

第 5 章: 代码组合（compose）

函数饲养

这就是 `组合` （compose，以下将称之为组合）：

```
var compose = function(f,g) {
  return function(x) {
    return f(g(x));
  };
};
```

`f` 和 `g` 都是函数，`x` 是在它们之间通过“管道”传输的值。

`组合` 看起来像是在饲养函数。你就是饲养员，选择两个有特点又遭你喜欢的函数，让它们结合，产下一个崭新的函数。组合的用法如下：

```
var toUpperCase = function(x) { return x.toUpperCase(); };
var exclaim = function(x) { return x + '!'; };
var shout = compose(exclaim, toUpperCase);

shout("send in the clowns");
//=> "SEND IN THE CLOWNS!"
```

两个函数组合之后返回了一个新函数是完全讲得通的：组合某种类型（本例中是函数）的两个元素本就该生成一个该类型的新元素。把两个乐高积木组合起来绝不可能得到一个林肯积木。所以这是有道理的，我们将在适当的时候探讨这方面的一些底层理论。

在 `compose` 的定义中，`g` 将先于 `f` 执行，因此就创建了一个从右到左的数据流。这样做的可读性远远高于嵌套一大堆的函数调用，如果不用组合，`shout` 函数将会是这样的：

```
var shout = function(x){
  return exclaim(toUpperCase(x));
};
```

让代码从右向左运行，而不是由内而外运行，我觉得可以称之为“左倾”（吁——）。我们来看一个顺序很重要的例子：

```
var head = function(x) { return x[0]; };
var reverse = reduce(function(acc, x){ return [x].concat(acc); }
, []);
var last = compose(head, reverse);

last(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'uppercut'
```

`reverse` 反转列表， `head` 取列表中的第一个元素；所以结果就是得到了一个 `last` 函数（译者注：即取列表的最后一个元素），虽然它性能不高。这个组合中函数的执行顺序应该是显而易见的。尽管我们可以定义一个从左向右的版本，但是从右向左执行更加能够反映数学上的含义——是的，组合的概念直接来自于数学课本。实际上，现在是时候去看看所有的组合都有的一个特性了。

```
// 结合律（associativity）
var associative = compose(f, compose(g, h)) == compose(compose(f
, g), h);
// true
```

这个特性就是结合律，符合结合律意味着不管你是把 `g` 和 `h` 分到一组，还是把 `f` 和 `g` 分到一组都不重要。所以，如果我们想把字符串变为大写，可以这么写：

```
compose(toUpperCase, compose(head, reverse));

// 或者
compose(compose(toUpperCase, head), reverse);
```

因为如何为 `compose` 的调用分组不重要，所以结果都是一样的。这也让我们有能力写一个可变的组合（variadic compose），用法如下：

```
// 前面的例子中我们必须要写两个组合才行，但既然组合是符合结合律的，我们就可以只写一个，
// 而且想传给它多少个函数就传给它多少个，然后让它自己决定如何分组。

var lastUpper = compose(toUpperCase, head, reverse);

lastUpper(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'UPPERCUT'


var loudLastUpper = compose(exclaim, toUpperCase, head, reverse)

loudLastUpper(['jumpkick', 'roundhouse', 'uppercut']);
//=> 'UPPERCUT!'
```

运用结合律能为我们带来强大的灵活性，还有对执行结果不会出现意外的那种平和心态。至于稍微复杂些的可变组合，也都包含在本书的 `support` 库里了，而且你也可以在类似 lodash、underscore 以及 ramda 这样的类库中找到它们的常规定义。

结合律的一大好处是任何一个函数分组都可以被拆开来，然后再以它们自己的组合方式打包在一起。让我们来重构重构前面的例子：

```
var loudLastUpper = compose(exclaim, toUpperCase, head, reverse)
;

// 或
var last = compose(head, reverse);
var loudLastUpper = compose(exclaim, toUpperCase, last);

// 或
var last = compose(head, reverse);
var angry = compose(exclaim, toUpperCase);
var loudLastUpper = compose(angry, last);

// 更多变种...
```

关于如何组合，并没有标准的答案——我们只是以自己喜欢的方式搭乐高积木罢了。通常来说，最佳实践是让组合可重用，就像 `last` 和 `angry` 那样。如果熟悉 Fowler 的《重构》一书的话，你可能会认识到这个过程叫做 “extract method”——只不过不需要关心对象的状态。

pointfree

pointfree 模式指的是，永远不必说出你的数据。咳咳对不起（译者注：此处原文是“Pointfree style means never having to say your data”，源自 1970 年的电影 *Love Story* 里的一句著名台词“Love means never having to say you're sorry”。紧接着作者又说了一句“Excuse me”，大概是一种幽默）。它的意思是说，函数无须提及将要操作的数据是什么样的。一等公民的函数、柯里化（curry）以及组合协作起来非常有助于实现这种模式。

```
// 非 pointfree，因为提到了数据：word
var snakeCase = function (word) {
  return word.toLowerCase().replace(/\s+/ig, '_');
};

// pointfree
var snakeCase = compose(replace(/\s+/ig, '_'), toLowerCase);
```

看到 `replace` 是如何被局部调用的了么？这里所做的事情就是通过管道把数据在接受单个参数的函数间传递。利用 curry，我们能够做到让每个函数都先接收数据，然后操作数据，最后再把数据传递到下一个函数那里去。另外注意在 pointfree 版本中，不需要 `word` 参数就能构造函数；而在非 pointfree 的版本中，必须要有 `word` 才能进行一切操作。

我们再来看一个例子。

```
// 非 pointfree，因为提到了数据：name
var initials = function (name) {
  return name.split(' ').map(compose(toUpperCase, head)).join('. ');
};

// pointfree
var initials = compose(join('. '), map(compose(toUpperCase, head)), split(' '));

initials("hunter stockton thompson");
// 'H. S. T'
```

另外，pointfree 模式能够帮助我们减少不必要的命名，让代码保持简洁和通用。对函数式代码来说，pointfree 是非常好的石蕊试验，因为它能告诉我们一个函数是否是接受输入返回输出的小函数。比如，while 循环是不能组合的。不过你也要警惕，pointfree 就像是一把双刃剑，有时候也能混淆视听。并非所有的函数式代码都是 pointfree 的，不过这没关系。可以使用它的时候就使用，不能使用的时候就用普通函数。

debug

组合的一个常见错误是，在没有局部调用之前，就组合类似 `map` 这样接受两个参数的函数。

```
// 错误做法：我们传给了 `angry` 一个数组，根本不知道最后传给 `map` 的是什
么东西。
var latin = compose(map, angry, reverse);

latin(["frog", "eyes"]);
// error


// 正确做法：每个函数都接受一个实际参数。
var latin = compose(map(angry), reverse);

latin(["frog", "eyes"]);
// ["EYES!", "FROG!"])
```

如果在 debug 组合的时候遇到了困难，那么可以使用下面这个实用的，但是不纯的 `trace` 函数来追踪代码的执行情况。

```
var trace = curry(function(tag, x){
  console.log(tag, x);
  return x;
});

var dasherize = compose(join('-'), toLower, split(' '), replace(
/\s{2,}/ig, ' '));

dasherize('The world is a vampire');
// TypeError: Cannot read property 'apply' of undefined
```

这里报错了，来 `trace` 下：

```
var dasherize = compose(join('-'), toLower, trace("after split")
, split(' '), replace(/\s{2,}/ig, ' '));
// after split [ 'The', 'world', 'is', 'a', 'vampire' ]
```

啊！ `toLower` 的参数是一个数组，所以需要先用 `map` 调用一下它。

```
var dasherize = compose(join('-'), map(toLower), split(' '), rep
lace(/\s{2,}/ig, ' '));

dasherize('The world is a vampire');

// 'the-world-is-a-vampire'
```

`trace` 函数允许我们在某个特定的点观察数据以便 debug。像 haskell 和 purescript 之类的语言出于开发的方便，也都提供了类似的函数。

组合将成为我们构造程序的工具，而且幸运的是，它背后是有一个强大的理论做支撑的。让我们来研究研究这个理论。

范畴学

范畴学（category theory）是数学中的一个抽象分支，能够形式化诸如集合论（set theory）、类型论（type theory）、群论（group theory）以及逻辑学（logic）等数学分支中的一些概念。范畴学主要处理对象（object）、态射（morphism）和变化式（transformation），而这些概念跟编程的联系非常紧密。下图是一些相同的概念分别在不同理论下的形式：

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{ (x, x) \mid x \in A \}$	path space A^I

抱歉，我没有任何要吓唬你的意思。我并不假设你对这些概念都了如指掌，我只是想让你明白这里面有多少重复的内容，让你知道为何范畴学要统一这些概念。

在范畴学中，有一个概念叫做...范畴。有着以下这些组件（component）的搜集（collection）就构成了一个范畴：

- 对象的搜集
- 态射的搜集
- 态射的组合
- identity 这个独特的态射

范畴学抽象到足以模拟任何事物，不过目前我们最关心的还是类型和函数，所以让我们把范畴学运用到它们身上看看。

对象的搜集

对象就是数据类型，例如 `String` 、 `Boolean` 、 `Number` 和 `Object` 等等。通常我们把数据类型视作所有可能的值的一个集合（set）。像 `Boolean` 就可以看作是 `[true, false]` 的集合， `Number` 可以是所有实数的一个集合。把类型当作集合对待是有好处的，因为我们可以利用集合论（set theory）处理类型。

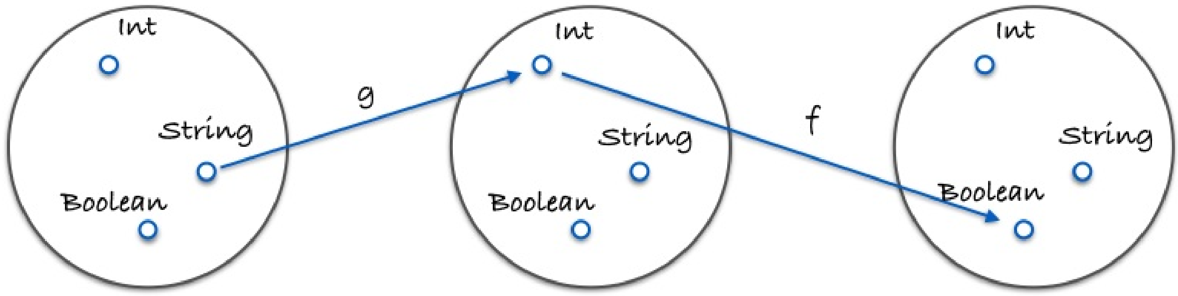
态射的搜集

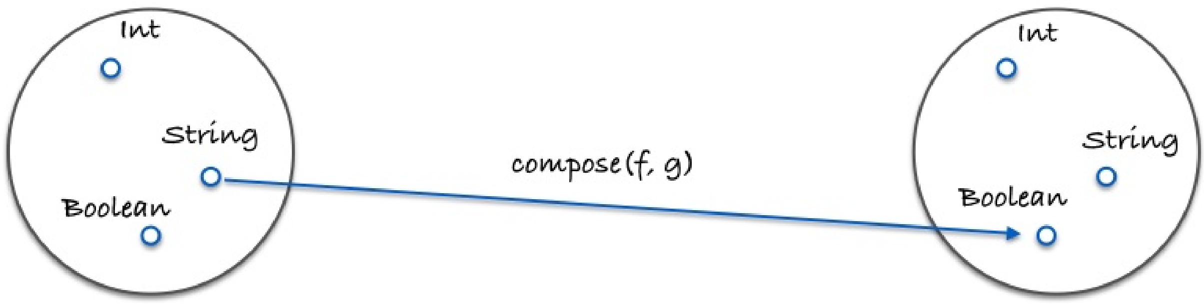
态射是标准的、普通的纯函数。

态射的组合

你可能猜到了，这就是本章介绍的新玩意儿—— `组合` 。我们已经讨论过 `compose` 函数是符合结合律的，这并非巧合，结合律是在范畴学中对任何组合都适用的一个特性。

这张图展示了什么是组合：





这里有一个具体的例子：

```
var g = function(x){ return x.length; };
var f = function(x){ return x === 4; };
var isFourLetterWord = compose(f, g);
```

identity 这个独特的态射

让我们介绍一个名为 `id` 的实用函数。这个函数接受随便什么输入然后原封不动地返回它：

```
var id = function(x){ return x; };
```

你可能会问“这到底哪里有用了？”。别急，我们会在随后的章节中拓展这个函数的，暂时先把它当作一个可以替代给定值的函数——一个假装自己是普通数据的函数。

`id` 函数跟组合一起使用简直完美。下面这个特性对所有的一元函数（unary function）（一元函数：只接受一个参数的函数） `f` 都成立：

```
// identity
compose(id, f) == compose(f, id) == f;
// true
```

嘿，这就是实数的单位元（identity property）嘛！如果这还不够清楚直白，别着急，慢慢理解它的无用性。很快我们就会到处使用 `id` 了，不过暂时我们还是把它当作一个替代给定值的函数。这对写 pointfree 的代码非常有用。

好了，以上就是类型和函数的范畴。不过如果你是第一次听说这些概念，我估计你还是有些迷糊，不知道范畴到底是什么，为什么有用。没关系，本书全书都在借助这些知识编写示例代码。至于现在，就在本章，本行文字中，你至少可以认为它向我们提供了有关组合的知识——比如结合律和单位律。

除了类型和函数，还有什么范畴呢？还有很多，比如我们可以定义一个有向图（directed graph），以节点为对象，以边为态射，以路径连接为组合。还可以定义一个实数类型（Number），以所有的实数为对象，以 `>=` 为态射（实际上任何偏序（partial order）或全序（total order）都可以成为一个范畴）。范畴的总数是无限的，但是要达到本书的目的，我们只需要关心上面定义的范畴就好了。至此我们已经大致浏览了一些表面的东西，必须要继续后面的内容了。

总结

组合像一系列管道那样把不同的函数联系在一起，数据就可以也必须在其中流动——毕竟纯函数就是输入对输出，所以打破这个链条就是不尊重输出，就会让我们的应用一无是处。

我们认为组合是高于其他所有原则的设计原则，这是因为组合让我们的代码简单而富有可读性。另外范畴学将在应用架构、模拟副作用和保证正确性方面扮演重要角色。

现在我们已经有足够的知识去进行一些实际的练习了，让我们来编写一个示例应用。

第 6 章: 示例应用

练习

```
require('../../support');
var _ = require('ramda');
var accounting = require('accounting');

// 示例数据
var CARS = [
    {name: "Ferrari FF", horsepower: 660, dollar_value: 700000,
in_stock: true},
    {name: "Spyker C12 Zagato", horsepower: 650, dollar_value: 6
48000, in_stock: false},
    {name: "Jaguar XKR-S", horsepower: 550, dollar_value: 132000
, in_stock: false},
    {name: "Audi R8", horsepower: 525, dollar_value: 114200, in_
stock: false},
    {name: "Aston Martin One-77", horsepower: 750, dollar_value:
1850000, in_stock: true},
    {name: "Pagani Huayra", horsepower: 700, dollar_value: 13000
00, in_stock: false}
  ];

// 练习 1:
// ============
// 使用 _.compose() 重写下面这个函数。提示：_.prop() 是 curry 函数
var isLastInStock = function(cars) {
  var last_car = _.last(cars);
  return _.prop('in_stock', last_car);
};

// 练习 2:
// ============
// 使用 _.compose()、_.prop() 和 _.head() 获取第一个 car 的 name
var nameOfFirstCar = undefined;


// 练习 3:
// ============
// 使用帮助函数 _average 重构 averageDollarValue 使之成为一个组合
var _average = function(xs) { return reduce(add, 0, xs) / xs.len
gth; }; // <- 无须改动
```

```
var averageDollarValue = function(cars) {
  var dollar_values = map(function(c) { return c.dollar_value; }
, cars);
  return _average(dollar_values);
};


// 练习 4:
// ============
// 使用 compose 写一个 sanitizeNames() 函数，返回一个下划线连接的小写字
符串：例如：sanitizeNames(["Hello World"]) //=> ["hello_world"]。

var _underscore = replace(/\W+/g, '_'); //<-- 无须改动，并在 sanit
izeNames 中使用它

var sanitizeNames = undefined;


// 彩蛋 1:
// ============
// 使用 compose 重构 availablePrices

var availablePrices = function(cars) {
  var available_cars = _.filter(_.prop('in_stock'), cars);
  return available_cars.map(function(x){
    return accounting.formatMoney(x.dollar_value);
  }).join(', ');
};


// 彩蛋 2:
// ============
// 重构使之成为 pointfree 函数。提示：可以使用 _.flip()

var fastestCar = function(cars) {
  var sorted = _.sortBy(function(car){ return car.horsepower },
cars);
  var fastest = _.last(sorted);
  return fastest.name + ' is the fastest';
};
```

第 6 章: 示例应用

声明式代码

我们要开始转变观念了，从本章开始，我们将不再指示计算机如何工作，而是指出我们明确希望得到的结果。我敢保证，这种做法与那种需要时刻关心所有细节的命令式编程相比，会让你轻松许多。

与命令式不同，声明式意味着我们要写表达式，而不是一步一步的指示。

以 SQL 为例，它就没有“先做这个，再做那个”的命令，有的只是一个指明我们想要从数据库取什么数据的表达式。至于如何取数据则是由它自己决定的。以后数据库升级也好，SQL 引擎优化也好，根本不需要更改查询语句。这是因为，有多种方式解析一个表达式并得到相同的结果。

对包括我在内的一些人来说，一开始是不太容易理解“声明式”这个概念的；所以让我们写几个例子找找感觉。

```
// 命令式
var makes = [];
for (i = 0; i < cars.length; i++) {
  makes.push(cars[i].make);
}


// 声明式
var makes = cars.map(function(car){ return car.make; });
```

命令式的循环要求你必须先实例化一个数组，而且执行完这个实例化语句之后，解释器才继续执行后面的代码。然后再直接迭代 `cars` 列表，手动增加计数器，把各种零零散散的东西都展示出来...实在是直白得有些露骨。

使用 `map` 的版本是一个表达式，它对执行顺序没有要求。而且， `map` 函数如何进行迭代，返回的数组如何收集，都有很大的自由度。它指明的是 `做什么` ，不是 `怎么做` 。因此，它是正儿八经的声明式代码。

除了更加清晰和简洁之外， `map` 函数还可以进一步优化，这么一来我们宝贵的应用代码就无须改动了。

至于那些说“虽然如此，但使用命令式循环速度要快很多”的人，我建议你们先去学学 JIT 优化代码的相关知识。这里有一个非常棒的视频，可能会对你有帮助。

再看一个例子。

```
// 命令式
var authenticate = function(form) {
  var user = toUser(form);
  return logIn(user);
};

// 声明式
var authenticate = compose(logIn, toUser);
```

虽然命令式的版本并不一定就是错的，但还是硬编码了那种一步接一步的执行方式。而 `compose` 表达式只是简单地指出了这样一个事实：用户验证是 `toUser` 和 `logIn` 两个行为的组合。这再次说明，声明式为潜在的代码更新提供了支持，使得我们的应用代码成为了一种高级规范（high level specification）。

因为声明式代码不指定执行顺序，所以它天然地适合进行并行运算。它与纯函数一起解释了为何函数式编程是未来并行计算的一个不错选择——我们真的不需要做什么就能实现一个并行／并发系统。

一个函数式的 flickr

现在我们以一种声明式的、可组合的方式创建一个示例应用。暂时我们还是会作点小弊，使用副作用；但我们会把副作用的程度降到最低，让它们与纯函数代码分离开来。这个示例应用是一个浏览器 widget，功能是从 flickr 获取图片并在页面上展示。我们从写 html 开始：

```
<!DOCTYPE html>
<html>
  <head>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/require.js/2.1.11/require.min.js"></script>
    <script src="flickr.js"></script>
  </head>
  <body></body>
</html>
```

flickr.js 如下：

```
requirejs.config({
  paths: {
    ramda: 'https://cdnjs.cloudflare.com/ajax/libs/ramda/0.13.0/ramda.min',
    jquery: 'https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min'
  }
});

require([
    'ramda',
    'jquery'
  ],
  function (_, $) {
    var trace = _.curry(function(tag, x) {
      console.log(tag, x);
      return x;
    });
    // app goes here
  });
```

这里我们使用了 ramda ，没有用 lodash 或者其他类库。ramda 提供了 `compose` 、 `curry` 等很多函数。模块加载我们选择的是 requirejs，我以前用过 requirejs，虽然它有些重，但为了保持一致性，本书将一直使用它。另外，我也把 `trace` 函数写好了，便于 debug。

有点跑题了。言归正传，我们的应用将做 4 件事：

1. 根据特定搜索关键字构造 url
2. 向 flickr 发送 api 请求
3. 把返回的 json 转为 html 图片
4. 把图片放到屏幕上

注意到没？上面提到了两个不纯的动作，即从 flickr 的 api 获取数据和在屏幕上放置图片这两件事。我们先来定义这两个动作，这样就能隔离它们了。

```
var Impure = {
  getJSON: _.curry(function(callback, url) {
    $.getJSON(url, callback);
  }),

  setHtml: _.curry(function(sel, html) {
    $(sel).html(html);
  })
};
```

这里只是简单地包装了一下 jQuery 的 `getJSON` 方法，把它变为一个 curry 函数，还有就是把参数位置也调换了下。这些方法都在 `Impure` 命名空间下，这样我们就知道它们都是危险函数。在后面的例子中，我们会把这两个函数变纯。

下一步是构造 url 传给 `Impure.getJSON` 函数。

```
var url = function (term) {
  return 'https://api.flickr.com/services/feeds/photos_public.gne?tags=' + term + '&format=json&jsoncallback=?';
};
```

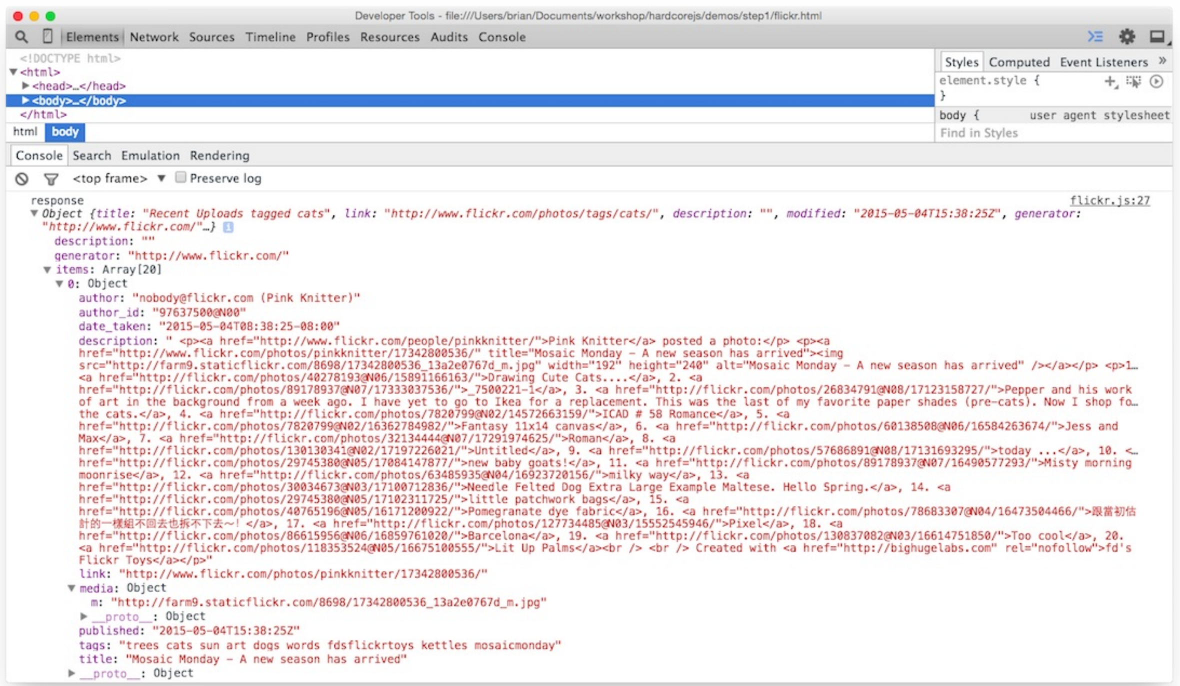
借助 monoid 或 combinator （后面会讲到这些概念），我们可以使用一些奇技淫巧来让 `url` 函数变为 pointfree 函数。但是为了可读性，我们还是选择以普通的非 pointfree 的方式拼接字符串。

让我们写一个 `app` 函数发送请求并把内容放置到屏幕上。

```
var app = _.compose(Impure.getJSON(trace("response")), url);

app("cats");
```

这会调用 `url` 函数，然后把字符串传给 `getJSON` 函数。`getJSON` 已经局部应用了 `trace`，加载这个应用将会把请求的响应显示在 console 里。



我们想要从这个 json 里构造图片，看起来 src 都在 `items` 数组中的每个 `media` 对象的 `m` 属性上。

不管怎样，我们可以使用 ramda 的一个通用 getter 函数 `_.prop()` 来获取这些嵌套的属性。不过为了让你明白这个函数做了什么事情，我们自己实现一个 prop 看看：

```
var prop = _.curry(function(property, object){
  return object[property];
});
```

实际上这有点傻，仅仅是用 `[]` 来获取一个对象的属性而已。让我们利用这个函数获取图片的 src。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var srcs = _.compose(_.map(mediaUrl), _.prop('items'));
```

一旦得到了 `items` ，就必须使用 `map` 来分解每一个 url；这样就得到了一个包含所有 src 的数组。把它和 `app` 联结起来，打印结果看看。

```
var renderImages = _.compose(Impure.setHtml("body"), srcs);
var app = _.compose(Impure.getJSON(renderImages), url);
```

这里所做的只不过是新建了一个组合，这个组合会调用 `srcs` 函数，并把返回结果设置为 body 的 html。我们也把 `trace` 替换为了 `renderImages` ，因为已经有了除原始 json 以外的数据。这将会粗暴地把所有的 src 直接显示在屏幕上。

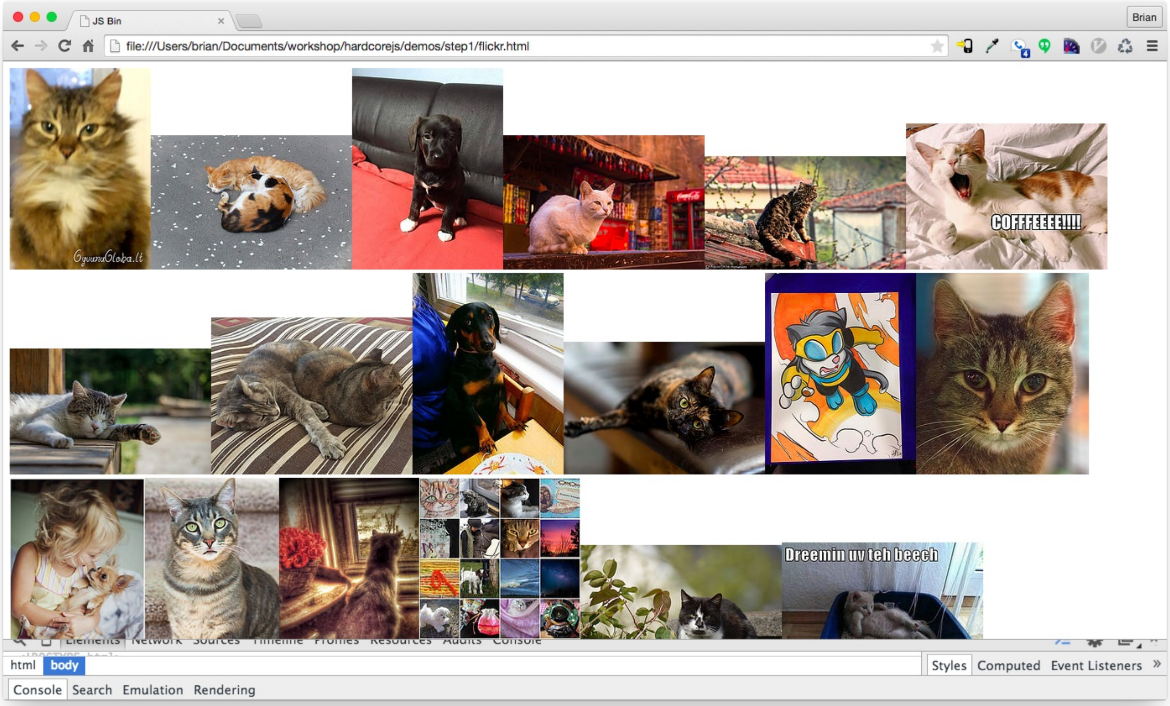
最后一步是把这些 src 变为真正的图片。对大型点的应用来说，是应该使用类似 Handlebars 或者 React 这样的 template/dom 库来做这件事的。但我们这个应用太小了，只需要一个 img 标签，所以用 jQuery 就好了。

```
var img = function (url) {
  return $('<img />', { src: url });
};
```

jQuery 的 `html()` 方法接受标签数组为参数，所以我们只须把 src 转换为 img 标签然后传给 `setHtml` 即可。

```
var images = _.compose(_.map(img), srcs);
var renderImages = _.compose(Impure.setHtml("body"), images);
var app = _.compose(Impure.getJSON(renderImages), url);
```

任务完成！



下面是完整代码：

```
requirejs.config({
  paths: {
    ramda: 'https://cdnjs.cloudflare.com/ajax/libs/ramda/0.13.0/
ramda.min',
    jquery: 'https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/
jquery.min'
  }
});

require([
    'ramda',
    'jquery'
  ],
  function (_, $) {
    ////////////////////////////////////////////
    // Utils

    var Impure = {
      getJSON: _.curry(function(callback, url) {
        $.getJSON(url, callback);
      }),
```

```
      setHtml: _.curry(function(sel, html) {
        $(sel).html(html);
      })
    };

    var img = function (url) {
      return $('<img />', { src: url });
    };

    var trace = _.curry(function(tag, x) {
      console.log(tag, x);
      return x;
    });

    ////////////////////////////////////////////

    var url = function (t) {
      return 'https://api.flickr.com/services/feeds/photos_publi
c.gne?tags=' + t + '&format=json&jsoncallback=?';
    };

    var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

    var srcs = _.compose(_.map(mediaUrl), _.prop('items'));

    var images = _.compose(_.map(img), srcs);

    var renderImages = _.compose(Impure.setHtml("body"), images)
;

    var app = _.compose(Impure.getJSON(renderImages), url);

    app("cats");
  });
```

看看，多么美妙的声明式规范啊，只说做什么，不说怎么做。现在我们可以把每一行代码都视作一个等式，变量名所代表的属性就是等式的含义。我们可以利用这些属性去推导分析和重构这个应用。

有原则的重构

上面的代码是有优化空间的——我们获取 url map 了一次，把这些 url 变为 img 标签又 map 了一次。关于 map 和组合是有定律的：

```
// map 的组合律
var law = compose(map(f), map(g)) == map(compose(f, g));
```

我们可以利用这个定律优化代码，进行一次有原则的重构。

```
// 原有代码
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var srcs = _.compose(_.map(mediaUrl), _.prop('items'));

var images = _.compose(_.map(img), srcs);
```

感谢等式推导（equational reasoning）及纯函数的特性，我们可以内联调用 `srcs` 和 `images`，也就是把 map 调用排列起来。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var images = _.compose(_.map(img), _.map(mediaUrl), _.prop('items'));
```

把 `map` 排成一列之后就可以应用组合律了。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var images = _.compose(_.map(_.compose(img, mediaUrl)), _.prop('items'));
```

现在只需要循环一次就可以把每一个对象都转为 img 标签了。我们把 map 调用的 compose 取出来放到外面，提高一下可读性。

```
var mediaUrl = _.compose(_.prop('m'), _.prop('media'));

var mediaToImg = _.compose(img, mediaUrl);

var images = _.compose(_.map(mediaToImg), _.prop('items'));
```

总结

我们已经见识到如何在一个小而不失真实的应用中运用新技能了，也已经使用过函数式这个“数学框架”来推导和重构代码了。但是异常处理以及代码分支呢？如何让整个应用都是函数式的，而不仅仅是把破坏性的函数放到命名空间下？如何让应用更安全更富有表现力？这些都是本书第 2 部分将要解决的问题。

第 7 章: Hindley-Milner 类型签名

Hindley-Milner 类型签名

初识类型

刚接触函数式编程的人很容易深陷类型签名（type signatures）的泥淖。类型（type）是让所有不同背景的人都能高效沟通的元语言。很大程度上，类型签名是以 “Hindley-Milner” 系统写就的，本章我们将一起探究下这个系统。

类型签名在写纯函数时所起的作用非常大，大到英语都不能望其项背。这些签名轻轻诉说着函数最不可告人的秘密。短短一行，就能暴露函数的行为和目的。类型签名还衍生出了 “自由定理（free theorems）” 的概念。因为类型是可以推断的，所以明确的类型签名并不是必要的；不过你完全可以写精确度很高的类型签名，也可以让它们保持通用、抽象。类型签名不但可以用于编译时检测（compile time checks），还是最好的文档。所以类型签名在函数式编程中扮演着非常重要的角色——重要程度远远超出你的想象。

JavaScript 是一种动态类型语言，但这并不意味着要一味否定类型。我们还是要和字符串、数值、布尔值等等类型打交道的；只不过，语言层面上没有相关的集成让我们时刻谨记各种数据的类型罢了。别担心，既然我们可以用类型签名生成文档，也可以用注释来达到区分类型的目的。

JavaScript 也有一些类型检查工具，比如 Flow，或者它的静态类型方言 TypeScript 。由于本书的目标是让读者能够熟练使用各种工具去书写函数式代码，所以我们将选择所有函数式语言都遵循的标准类型系统。

神秘的传奇故事

从积尘已久的数学书，到浩如烟海的学术论文；从每周必读的博客文章，到源代码本身，我们都能发现 Hindley-Milner 类型签名的身影。Hindley-Milner 并不是一个复杂的系统，但还是需要一些解释和练习才能完全掌握这个小型语言的要义。

```
//  capitalize :: String -> String
var capitalize = function(s){
  return toUpperCase(head(s)) + toLowerCase(tail(s));
}

capitalize("smurf");
//=> "Smurf"
```

这里，`capitalize` 接受一个 `String` 并返回了一个 `String`。先别管实现，我们感兴趣的是它的类型签名。

在 Hindley-Milner 系统中，函数都写成类似 `a -> b` 这个样子，其中 `a` 和 `b` 是任意类型的变量。因此，`capitalize` 函数的类型签名可以理解为“一个接受 `String` 返回 `String` 的函数”。换句话说，它接受一个 `String` 类型作为输入，并返回一个 `String` 类型的输出。

再来看一些函数签名：

```
//  strLength :: String -> Number
var strLength = function(s){
  return s.length;
}

//  join :: String -> [String] -> String
var join = curry(function(what, xs){
  return xs.join(what);
});

//  match :: Regex -> String -> [String]
var match = curry(function(reg, s){
  return s.match(reg);
});

//  replace :: Regex -> String -> String -> String
var replace = curry(function(reg, sub, s){
  return s.replace(reg, sub);
});
```

`strLength` 和 `capitalize` 类似：接受一个 `String` 然后返回一个 `Number`。

至于其他的，第一眼看起来可能会比较疑惑。不过在还不完全了解细节的情况下，你尽可以把最后一个类型视作返回值。那么 `match` 函数就可以这么理解：它接受一个 `Regex` 和一个 `String`，返回一个 `[String]`。但是，这里有一个非常有趣的地方，请允许我稍作解释。

对于 `match` 函数，我们完全可以把它的类型签名这样分组：

```
//  match :: Regex -> (String -> [String])
var match = curry(function(reg, s){
  return s.match(reg);
});
```

是的，把最后两个类型包在括号里就能反映更多的信息了。现在我们可以看出 `match` 这个函数接受一个 `Regex` 作为参数，返回一个从 `String` 到 `[String]` 的函数。因为 curry，造成的结果就是这样：给 `match` 函数一个 `Regex`，得到一个新函数，能够处理其 `String` 参数。当然了，我们并非一定要这么看待这个过程，但这样思考有助于理解为何最后一个类型是返回值。

```
//  match :: Regex -> (String -> [String])

//  onHoliday :: String -> [String]
var onHoliday = match(/holiday/ig);
```

每传一个参数，就会弹出类型签名最前面的那个类型。所以 `onHoliday` 就是已经有了 `Regex` 参数的 `match`。

```
//  replace :: Regex -> (String -> (String -> String))
var replace = curry(function(reg, sub, s){
  return s.replace(reg, sub);
});
```

但是在这段代码中，就像你看到的那样，为 `replace` 加上这么多括号未免有些多余。所以这里的括号是完全可以省略的，如果我们愿意，可以一次性把所有的参数都传进来；所以，一种更简单的思路是：`replace` 接受三个参数，分别是

`Regex` 、 `String` 和另一个 `String` ，返回的还是一个 `String` 。

最后几点：

```
//  id :: a -> a
var id = function(x){ return x; }

//  map :: (a -> b) -> [a] -> [b]
var map = curry(function(f, xs){
  return xs.map(f);
});
```

这里的 `id` 函数接受任意类型的 `a` 并返回同一个类型的数据。和普通代码一样，我们也可以在类型签名中使用变量。把变量命名为 `a` 和 `b` 只是一种约定俗成的习惯，你可以使用任何你喜欢的名称。对于相同的变量名，其类型也一定相同。这是非常重要的一个原则，所以我们必须重申： `a -> b` 可以是从任意类型的 `a` 到任意类型的 `b` ，但是 `a -> a` 必须是同一个类型。例如， `id` 可以是 `String -> String` ，也可以是 `Number -> Number` ，但不能是 `String -> Bool` 。

相似地， `map` 也使用了变量，只不过这里的 `b` 可能与 `a` 类型相同，也可能不相同。我们可以这么理解： `map` 接受两个参数，第一个是从任意类型 `a` 到任意类型 `b` 的函数；第二个是一个数组，元素是任意类型的 `a` ； `map` 最后返回的是一个类型 `b` 的数组。

类型签名的美妙令人印象深刻，希望你已经被它深深折服。类型签名简直能够一字一句地告诉我们函数做了什么事情。比如 `map` 函数就是这样：给定一个从 `a` 到 `b` 的函数和一个 `a` 类型的数组作为参数，它就能返回一个 `b` 类型的数组。 `map` 唯一的明智之举就是使用其函数参数调用每一个 `a` ，其他所有操作都是噱头。

辨别类型和它们的含义是一项重要的技能，这项技能可以让你在函数式编程的路上走得更远。不仅论文、博客和文档等更易理解，类型签名本身也基本上能够告诉你它的函数性（functionality）。要成为一个能够熟练读懂类型签名的人，你得勤于练习；不过一旦掌握了这项技能，你将会受益无穷，不读手册也能获取大量信息。

这里还有一些例子，你可以自己试试看能不能理解它们。

```
//  head :: [a] -> a
var head = function(xs){ return xs[0]; }

//  filter :: (a -> Bool) -> [a] -> [a]
var filter = curry(function(f, xs){
  return xs.filter(f);
});

//  reduce :: (b -> a -> b) -> b -> [a] -> b
var reduce = curry(function(f, x, xs){
  return xs.reduce(f, x);
});
```

`reduce` 可能是以上签名里让人印象最为深刻的一个，同时也是最复杂的一个了，所以如果你理解起来有困难的话，也不必气馁。为了满足你的好奇心，我还是试着解释一下吧；尽管我的解释远远不如你自己通过类型签名理解其含义来得有教益。

不保证解释完全正确...（译者注：此处原文是“here goes nothing”，一般用于人们在做没有把握的事情之前说的话。）注意看 `reduce` 的签名，可以看到它的第一个参数是个函数，这个函数接受一个 `b` 和一个 `a` 并返回一个 `b` 。那么这些 `a` 和 `b` 是从哪来的呢？很简单，签名中的第二个和第三个参数就是 `b` 和元素为 `a` 的数组，所以唯一合理的假设就是这里的 `b` 和每一个 `a` 都将传给前面说的函数作为参数。我们还可以看到， `reduce` 函数最后返回的结果是一个 `b` ，也就是说， `reduce` 的第一个参数函数的输出就是 `reduce` 函数的输出。知道了 `reduce` 的含义，我们才敢说上面关于类型签名的推理是正确的。

缩小可能性范围

一旦引入一个类型变量，就会出现一个奇怪的特性叫做 *parametricity*（http://en.wikipedia.org/wiki/Parametricity ）。这个特性表明，函数将会以一种统一的行为作用于所有的类型。我们来研究下：

```
// head :: [a] -> a
```

注意看 `head`，可以看到它接受 `[a]` 返回 `a`。我们除了知道参数是个`数组`，其他的一概不知；所以函数的功能就只限于操作这个数组上。在它对 `a` 一无所知的情况下，它可能对 `a` 做什么操作呢？换句话说，`a` 告诉我们它不是一个`特定`的类型，这意味着它可以是`任意`类型；那么我们的函数对每一个可能的类型的操作都必须保持统一。这就是 *parametricity* 的含义。要让我们来猜测 `head` 的实现的话，唯一合理的推断就是它返回数组的第一个，或者最后一个，或者某个随机的元素；当然，`head` 这个命名应该能给我们一些线索。

再看一个例子：

```
// reverse :: [a] -> [a]
```

仅从类型签名来看，`reverse` 可能的目的是什么？再次强调，它不能对 `a` 做任何特定的事情。它不能把 `a` 变成另一个类型，或者引入一个 `b`；这都是不可能的。那它可以排序么？答案是不能，没有足够的信息让它去为每一个可能的类型排序。它能重新排列么？可以的，我觉得它可以，但它必须以一种可预料的方式达成目标。另外，它也有可能删除或者重复某一个元素。重点是，不管在哪种情况下，类型 `a` 的多态性（polymorphism）都会大幅缩小 `reverse` 函数可能的行为的范围。

这种“可能性范围的缩小”（narrowing of possibility）允许我们利用类似 Hoogle 这样的类型签名搜索引擎去搜索我们想要的函数。类型签名所能包含的信息量真的非常大。

自由定理

类型签名除了能够帮助我们推断函数可能的实现，还能够给我们带来自由定理（free theorems）。下面是两个直接从 Wadler 关于此主题的论文 中随机选择的例子：

```
// head :: [a] -> a
compose(f, head) == compose(head, map(f));

// filter :: (a -> Bool) -> [a] -> [a]
compose(map(f), filter(compose(p, f))) == compose(filter(p), map
(f));
```

不用写一行代码你也能理解这些定理，它们直接来自于类型本身。第一个例子中，等式左边说的是，先获取数组的 `头部`（译者注：即第一个元素），然后对它调用函数 `f`；等式右边说的是，先对数组中的每一个元素调用 `f`，然后再取其返回结果的 `头部`。这两个表达式的作用是相等的，但是前者要快得多。

你可能会想，这不是常识么。但根据我的调查，计算机是没有常识的。实际上，计算机必须要有一种形式化方法来自动进行类似的代码优化。数学提供了这种方法，能够形式化直观的感觉，这无疑对死板的计算机逻辑非常有用。

第二个例子 `filter` 也是一样。等式左边是说，先组合 `f` 和 `p` 检查哪些元素要过滤掉，然后再通过 `map` 实际调用 `f`（别忘了 `filter` 是不会改变数组中元素的，这就保证了 `a` 将保持不变）；等式右边是说，先用 `map` 调用 `f`，然后再根据 `p` 过滤元素。这两者也是相等的。

以上只是两个例子，但它们传达的定理却是普适的，可以应用到所有的多态性类型签名上。在 JavaScript 中，你可以借助一些工具来声明重写规则，也可以直接使用 `compose` 函数来定义重写规则。总之，这么做的好处是显而易见且唾手可得的，可能性则是无限的。

类型约束

最后要注意的一点是，签名也可以把类型约束为一个特定的接口（interface）。

```
// sort :: Ord a => [a] -> [a]
```

胖箭头左边表明的是这样一个事实：`a` 一定是个 `Ord` 对象。也就是说，`a` 必须要实现 `Ord` 接口。`Ord` 到底是什么？它是从哪来的？在一门强类型语言中，它可能就是一个自定义的接口，能够让不同的值排序。通过这种方式，我们不仅能够获取关于 `a` 的更多信息，了解 `sort` 函数具体要干什么，而且还能限制函数的作用范围。我们把这种接口声明叫做类型约束（type constraints）。

```
// assertEqual :: (Eq a, Show a) => a -> a -> Assertion
```

这个例子中有两个约束：`Eq` 和 `Show`。它们保证了我们可以检查不同的 `a` 是否相等，并在有不相等的情况下打印出其中的差异。

我们将会在后面的章节中看到更多类型约束的例子，其含义也会更加清晰。

总结

Hindley-Milner 类型签名在函数式编程中无处不在，它们简单易读，写起来也不复杂。但仅仅凭签名就能理解整个程序还是有一定难度的，要想精通这个技能就更需要花点时间了。从这开始，我们将给每一行代码都加上类型签名。

第 8 章: 特百惠

特百惠

（译者注：特百惠是美国家居用品品牌，代表产品是塑料容器。）

强大的容器



我们已经知道如何书写函数式的程序了，即通过管道把数据在一系列纯函数间传递的程序。我们也知道了，这些程序就是声明式的行为规范。但是，控制流（control flow）、异常处理（error handling）、异步操作（asynchronous actions）和状态（state）呢？还有更棘手的作用（effects）呢？本章将对上述这些抽象概念赖以建立的基础作一番探究。

首先我们将创建一个容器（container）。这个容器必须能够装载任意类型的值；否则的话，像只能装木薯布丁的密封塑料袋是没什么用的。这个容器将会是一个对象，但我们不会为它添加面向对象观念下的属性和方法。是的，我们将把它当作一

个百宝箱——一个存放宝贵的数据的特殊盒子。

```
var Container = function(x) {
  this.__value = x;
}

Container.of = function(x) { return new Container(x); };
```

这是本书的第一个容器，我们贴心地把它命名为 `Container`。我们将使用 `Container.of` 作为构造器（constructor），这样就不用到处去写糟糕的 `new` 关键字了，非常省心。实际上不能这么简单地看待 `of` 函数，但暂时先认为它是把值放到容器里的一种方式。

我们来检验下这个崭新的盒子：

```
Container.of(3)
//=> Container(3)


Container.of("hotdogs")
//=> Container("hotdogs")


Container.of(Container.of({name: "yoda"}))
//=> Container(Container({name: "yoda" }))
```

如果用的是 node，那么你会看到打印出来的是 `{__value: x}`，而不是实际值 `Container(x)`；Chrome 打印出来的是正确的。不过这并不重要，只要你理解 `Container` 是什么样的就行了。有些环境下，你也可以重写 `inspect` 方法，但我们不打算涉及这方面的知识。在本书中，出于教学和美学上的考虑，我们将把概念性的输出都写成好像 `inspect` 被重写了的样子，因为这样写的教育意义将远远大于 `{__value: x}`。

在继续后面的内容之前，先澄清几点：

- `Container` 是个只有一个属性的对象。尽管容器可以有不止一个的属性，但大多数容器还是只有一个。我们很随意地把 `Container` 的这个属性命名为 `__value`。

- `__value` 不能是某个特定的类型，不然 `Container` 就对不起它这个名字了。
- 数据一旦存放到 `Container`，就会一直待在那儿。我们可以用 `.__value` 获取到数据，但这样做有悖初衷。

如果把容器想象成玻璃罐的话，上面这三条陈述的理由就会比较清晰了。但是暂时，请先保持耐心。

第一个 functor

一旦容器里有了值，不管这个值是什么，我们就需要一种方法来让别的函数能够操作它。

```
// (a -> b) -> Container a -> Container b
Container.prototype.map = function(f){
  return Container.of(f(this.__value))
}
```

这个 `map` 跟数组那个著名的 `map` 一样，除了前者的参数是 `Container a` 而后者是 `[a]`。它们的使用方式也几乎一致：

```
Container.of(2).map(function(two){ return two + 2 })
//=> Container(4)


Container.of("flamethrowers").map(function(s){ return s.toUpperCase() })
//=> Container("FLAMETHROWERS")


Container.of("bombs").map(concat(' away')).map(_.prop('length'))
//=> Container(10)
```

为什么要使用这样一种方法？因为我们能够在不离开 `Container` 的情况下操作容器里面的值。这是非常了不起的一件事情。`Container` 里的值传递给 `map` 函数之后，就可以任我们操作；操作结束后，为了防止意外再把它放回它所属的

`Container` 。这样做的结果是，我们能连续地调用 `map` ，运行任何我们想运行的函数。甚至还可以改变值的类型，就像上面最后一个例子中那样。

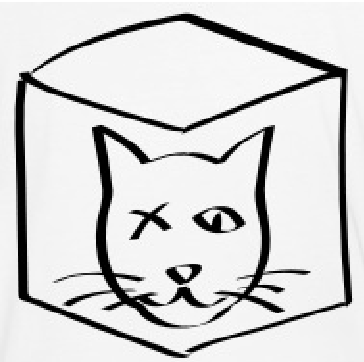
等等，如果我们能一直调用 `map` ，那它不就是个组合（composition）么！这里边是有什么数学魔法在起作用？是 *functor*。各位，这个数学魔法就是 *functor*。

> functor 是实现了 `map` 函数并遵守一些特定规则的容器类型。

没错，*functor* 就是一个签了合约的接口。我们本来可以简单地把它称为 `Mappable` ，但这样就没有 *fun*（译者注：指 `functor` 中包含 `fun` 这个单词，是一双关语）了，对吧？functor 是范畴学里的概念，我们将在本章末尾详细探索与此相关的数学知识；暂时我们先用这个名字很奇怪的接口做一些不那么理论的、实用性的练习。

把值装进一个容器，而且只能使用 `map` 来处理它，这么做的理由到底是什么呢？如果我们换种方式来问，答案就很明显了：让容器自己去运用函数能给我们带来什么好处？答案是抽象，对于函数运用的抽象。当 `map` 一个函数的时候，我们请求容器来运行这个函数。不夸张地讲，这是一种十分强大的理念。

薛定谔的 Maybe



说实话 `Container` 挺无聊的，而且通常我们称它为 `Identity` ，与 `id` 函数的作用相同（这里也是有数学上的联系的，我们会在适当时候加以说明）。除此之外，还有另外一种 functor，那就是实现了 `map` 函数的类似容器的数据类型，这种 functor 在调用 `map` 的时候能够提供非常有用的行为。现在让我们来定义一个这样的 functor。

```
var Maybe = function(x) {
  this.__value = x;
}

Maybe.of = function(x) {
  return new Maybe(x);
}

Maybe.prototype.isNothing = function() {
  return (this.__value === null || this.__value === undefined);
}

Maybe.prototype.map = function(f) {
  return this.isNothing() ? Maybe.of(null) : Maybe.of(f(this.__v
alue));
}
```

`Maybe` 看起来跟 `Container` 非常类似，但是有一点不同：`Maybe` 会先检查自己的值是否为空，然后才调用传进来的函数。这样我们在使用 `map` 的时候就能避免恼人的空值了（注意这个实现出于教学目的做了简化）。

```
Maybe.of("Malkovich Malkovich").map(match(/a/ig));
//=> Maybe(['a', 'a'])

Maybe.of(null).map(match(/a/ig));
//=> Maybe(null)

Maybe.of({name: "Boris"}).map(_.prop("age")).map(add(10));
//=> Maybe(null)

Maybe.of({name: "Dinah", age: 14}).map(_.prop("age")).map(add(10
));
//=> Maybe(24)
```

注意看，当传给 `map` 的值是 `null` 时，代码并没有爆出错误。这是因为每一次 `Maybe` 要调用函数的时候，都会先检查它自己的值是否为空。

这种点记法（dot notation syntax）已经足够函数式了，但是正如在第 1 部分指出的那样，我们更想保持一种 pointfree 的风格。碰巧的是， `map` 完全有能力以 curry 函数的方式来“代理”任何 functor：

```
//  map :: Functor f => (a -> b) -> f a -> f b
var map = curry(function(f, any_functor_at_all) {
  return any_functor_at_all.map(f);
});
```

这样我们就可以像平常一样使用组合，同时也能正常使用 `map` 了，非常振奋人心。ramda 的 `map` 也是这样。后面的章节中，我们将在点记法更有教育意义的时候使用点记法，在方便使用 pointfree 模式的时候就用 pointfree。你注意到了么？我在类型标签中偷偷引入了一个额外的标记： `Functor f =>` 。这个标记告诉我们 `f` 必须是一个 functor。没什么复杂的，但我觉得有必要提一下。

用例

实际当中， `Maybe` 最常用在那些可能会无法成功返回结果的函数中。

```
//  safeHead :: [a] -> Maybe(a)
var safeHead = function(xs) {
  return Maybe.of(xs[0]);
};

var streetName = compose(map(_.prop('street')), safeHead, _.prop(
'addresses'));

streetName({addresses: []});
// Maybe(null)

streetName({addresses: [{street: "Shady Ln.", number: 4201}]});
// Maybe("Shady Ln.")
```

`safeHead` 与一般的 `_.head` 类似，但是增加了类型安全保证。引入 `Maybe` 会发生一件非常有意思的事情，那就是我们被迫要与狡猾的 `null` 打交道了。 `safeHead` 函数能够诚实地预告它可能的失败——失败真没什么可耻的——

然后返回一个 `Maybe` 来通知我们相关信息。实际上不仅仅是通知，因为毕竟我们想要的值深藏在 `Maybe` 对象中，而且只能通过 `map` 来操作它。本质上，这是一种由 `safeHead` 强制执行的空值检查。有了这种检查，我们才能在夜里安然入睡，因为我们知道最不受人待见的 `null` 不会突然出现。类似这样的 API 能够把一个像纸糊起来的、脆弱的应用升级为实实在在的、健壮的应用，这样的 API 保证了更加安全的软件。

有时候函数可以明确返回一个 `Maybe(null)` 来表明失败，例如：

```
//  withdraw :: Number -> Account -> Maybe(Account)
var withdraw = curry(function(amount, account) {
  return account.balance >= amount ?
    Maybe.of({balance: account.balance - amount}) :
    Maybe.of(null);
});

//  finishTransaction :: Account -> String
var finishTransaction = compose(remainingBalance, updateLedger);
// <- 假定这两个函数已经在别处定义好了

//  getTwenty :: Account -> Maybe(String)
var getTwenty = compose(map(finishTransaction), withdraw(20));


getTwenty({ balance: 200.00});
// Maybe("Your balance is $180.00")

getTwenty({ balance: 10.00});
// Maybe(null)
```

要是钱不够，`withdraw` 就会对我们嗤之以鼻然后返回一个 `Maybe(null)` 。`withdraw` 也显示出了它的多变性，使得我们后续的操作只能用 `map` 来进行。这个例子与前面例子不同的地方在于，这里的 `null` 是有意的。我们不用 `Maybe(String)` ，而是用 `Maybe(null)` 来发送失败的信号，这样程序在收到信号后就能立刻停止执行。这一点很重要：如果 `withdraw` 失败

了， `map` 就会切断后续代码的执行，因为它根本就不会运行传递给它的函数，即 `finishTransaction` 。这正是预期的效果：如果取款失败，我们并不想更新或者显示账户余额。

释放容器里的值

人们经常忽略的一个事实是：任何事物都有个最终尽头。那些会产生作用的函数，不管它们是发送 JSON 数据，还是在屏幕上打印东西，还是更改文件系统，还是别的什么，都要有一个结束。但是我们无法通过 `return` 把输出传递到外部世界，必须要运行这样或那样的函数才能传递出去。关于这一点，可以借用禅宗公案的口吻来叙述："如果一个程序运行之后没有可观察到的作用，那它到底运行了没有？"。或者，运行之后达到自身的目的了没有？有可能它只是浪费了几个 CPU 周期然后就去睡觉了...

应用程序所做的工作就是获取、更改和保存数据直到不再需要它们，对数据做这些操作的函数有可能被 `map` 调用，这样的话数据就可以不用离开它温暖舒适的容器。讽刺的是，有一种常见的错误就是试图以各种方法删除 `Maybe` 里的值，好像这个不确定的值是魔鬼，删除它就能让它突然显形，然后一切罪恶都会得到宽恕似的（译者注：此处原文应该是源自圣经）。要知道，我们的值没有完成它的使命，很有可能是其他代码分支造成的。我们的代码，就像薛定谔的猫一样，在某个特定的时间点有两种状态，而且应该保持这种状况不变直到最后一个函数为止。这样，哪怕代码有很多逻辑性的分支，也能保证一种线性的工作流。

不过，对容器里的值来说，还是有个逃生口可以出去。也就是说，如果我们想返回一个自定义的值然后还能继续执行后面的代码的话，是可以做到的；要达到这一目的，可以借助一个帮助函数 `maybe` ：

```
//  maybe :: b -> (a -> b) -> Maybe a -> b
var maybe = curry(function(x, f, m) {
  return m.isNothing() ? x : f(m.__value);
});

//  getTwenty :: Account -> String
var getTwenty = compose(
  maybe("You're broke!", finishTransaction), withdraw(20)
);


getTwenty({ balance: 200.00});
// "Your balance is $180.00"

getTwenty({ balance: 10.00});
// "You're broke!"
```

这样就可以要么返回一个静态值（与 `finishTransaction` 返回值的类型一致），要么继续愉快地在没有 `Maybe` 的情况下完成交易。`maybe` 使我们得以避免普通 `map` 那种命令式的 `if/else` 语句：`if(x !== null) { return f(x) }`。

引入 `Maybe` 可能会在初期造成一些不适。Swift 和 Scala 用户知道我在说什么，因为这两门语言的核心库里就有 `Maybe` 的概念，只不过伪装成 `Option(al)` 罢了。被迫在任何情况下都进行空值检查（甚至有些时候我们可以确定某个值不会为空），的确让大部分人头疼不已。然而随着时间推移，空值检查会成为第二本能，说不定你还会感激它提供的安全性呢。不管怎么说，空值检查大多数时候都能防止在代码逻辑上偷工减料，让我们脱离危险。

编写不安全的软件就像用蜡笔小心翼翼地画彩蛋，画完之后把它们扔到大街上一样（译者注：意思是彩蛋非常易于寻找。来源于复活节习俗，人们会藏起一些彩蛋让孩子寻找），或者像用三只小猪警告过的材料盖个养老院一样（译者注：来源于“三只小猪”童话故事）。`Maybe` 能够非常有效地帮助我们增加函数的安全性。

有一点我必须要提及，否则就太不负责任了，那就是 `Maybe` 的“真正”实现会把它分为两种类型：一种是非空值，另一种是空值。这种实现允许我们遵守 `map` 的 parametricity 特性，因此 `null` 和 `undefined` 能够依然被 `map` 调用，

functor 里的值所需的那种普遍性条件也能得到满足。所以你会经常看到 `Some(x) / None` 或者 `Just(x) / Nothing` 这样的容器类型在做空值检查，而不是 `Maybe` 。

“纯”错误处理



说出来可能会让你震惊， `throw/catch` 并不十分“纯”。当一个错误抛出的时候，我们没有收到返回值，反而是得到了一个警告！抛错的函数吐出一大堆的 0 和 1 作为盾和矛来攻击我们，简直就像是在反击输入值的入侵而进行的一场电子大作战。有了 `Either` 这个新朋友，我们就能以一种比向输入值宣战好得多的方式来处理错误，那就是返回一条非常礼貌的消息作为回应。我们来看一下：

```
var Left = function(x) {
  this.__value = x;
}

Left.of = function(x) {
  return new Left(x);
}

Left.prototype.map = function(f) {
  return this;
}

var Right = function(x) {
  this.__value = x;
}

Right.of = function(x) {
  return new Right(x);
}

Right.prototype.map = function(f) {
  return Right.of(f(this.__value));
}
```

`Left` 和 `Right` 是我们称之为 `Either` 的抽象类型的两个子类。我略去了创建 `Either` 父类的繁文缛节，因为我们不会用到它的，但你了解一下也没坏处。注意看，这里除了有两个类型，没别的新鲜东西。来看看它们是怎么运行的：

```
Right.of("rain").map(function(str){ return "b"+str; });
// Right("brain")

Left.of("rain").map(function(str){ return "b"+str; });
// Left("rain")

Right.of({host: 'localhost', port: 80}).map(_.prop('host'));
// Right('localhost')

Left.of("rolls eyes...").map(_.prop("host"));
// Left('rolls eyes...')
```

`Left` 就像是青春期少年那样无视我们要 `map` 它的请求。`Right` 的作用就像是一个 `Container`（也就是 Identity）。这里强大的地方在于，`Left` 有能力在它内部嵌入一个错误消息。

假设有一个可能会失败的函数，就拿根据生日计算年龄来说好了。的确，我们可以用 `Maybe(null)` 来表示失败并把程序引向另一个分支，但是这并没有告诉我们太多信息。很有可能我们想知道失败的原因是什么。用 `Either` 写一个这样的程序看看：

```
var moment = require('moment');

//  getAge :: Date -> User -> Either(String, Number)
var getAge = curry(function(now, user) {
  var birthdate = moment(user.birthdate, 'YYYY-MM-DD');
  if(!birthdate.isValid()) return Left.of("Birth date could not
be parsed");
  return Right.of(now.diff(birthdate, 'years'));
});

getAge(moment(), {birthdate: '2005-12-12'});
// Right(9)

getAge(moment(), {birthdate: 'balloons!'});
// Left("Birth date could not be parsed")
```

这么一来，就像 `Maybe(null)` ，当返回一个 `Left` 的时候就直接让程序短路。跟 `Maybe(null)` 不同的是，现在我们对程序为何脱离原先轨道至少有了一点头绪。有一件事要注意，这里返回的是 `Either(String, Number)` ，意味着我们这个 `Either` 左边的值是 `String` ，右边（译者注：也就是正确的值）的值是 `Number` 。这个类型签名不是很正式，因为我们并没有定义一个真正的 `Either` 父类；但我们还是从这个类型那里了解到不少东西。它告诉我们，我们得到的要么是一条错误消息，要么就是正确的年龄值。

```
//  fortune :: Number -> String
var fortune  = compose(concat("If you survive, you will be "), add(1));

//  zoltar :: User -> Either(String, _)
var zoltar = compose(map(console.log), map(fortune), getAge(moment()));

zoltar({birthdate: '2005-12-12'});
// "If you survive, you will be 10"
// Right(undefined)

zoltar({birthdate: 'balloons!'});
// Left("Birth date could not be parsed")
```

如果 `birthdate` 合法，这个程序就会把它神秘的命运打印在屏幕上让我们见证；如果不合法，我们就会收到一个有着清清楚楚的错误消息的 `Left` ，尽管这个消息是稳稳当当地待在它的容器里的。这种行为就像，虽然我们在抛错，但是是以一种平静温和的方式抛错，而不是像一个小孩子那样，有什么不对劲就闹脾气大喊大叫。

在这个例子中，我们根据 `birthdate` 的合法性来控制代码的逻辑分支，同时又让代码进行从右到左的直线运动，而不用爬过各种条件语句的大括号。通常，我们不会把 `console.log` 放到 `zoltar` 函数里，而是在调用 `zoltar` 的时候才 `map` 它，不过本例中，让你看看 `Right` 分支如何与 `Left` 不同也是很有帮助的。我们在 `Right` 分支的类型签名中使用 `_` 表示一个应该忽略的值（在有些浏览器中，你必须要 `console.log.bind(console)` 才能把 `console.log` 当作一等公民使用）。

我想借此机会指出一件你可能没注意到的事：这个例子中，尽管 `fortune` 使用了 `Either`，它对每一个 functor 到底要干什么却是毫不知情的。前面例子中的 `finishTransaction` 也是一样。通俗点来讲，一个函数在调用的时候，如果被 `map` 包裹了，那么它就会从一个非 functor 函数转换为一个 functor 函数。我们把这个过程叫做 *lift*。一般情况下，普通函数更适合操作普通的数据类型而不是容器类型，在必要的时候再通过 *lift* 变为合适的容器去操作容器类型。这样做的好处是能得到更简单、重用性更高的函数，它们能够随需求而变，兼容任意 functor。

`Either` 并不仅仅只对合法性检查这种一般性的错误作用非凡，对一些更严重的、能够中断程序执行的错误比如文件丢失或者 socket 连接断开等，`Either` 同样效果显著。你可以试试把前面例子中的 `Maybe` 替换为 `Either`，看怎么得到更好的反馈。

此刻我忍不住在想，我仅仅是把 `Either` 当作一个错误消息的容器介绍给你！这样的介绍有失偏颇，它的能耐远不止于此。比如，它表示了逻辑或（也就是 `||`）。再比如，它体现了范畴学里 *coproduct* 的概念，当然本书不会涉及这方面的知识，但值得你去深入了解，因为这个概念有很多特性值得利用。还比如，它是标准的 sum type（或者叫不交并集，disjoint union of sets），因为它含有的所有可能的值的总数就是它包含的那两种类型的总数（我知道这么说你听不懂，没关系，这里有一篇非常棒的文章讲述这个问题）。`Either` 能做的事情多着呢，但是作为一个 functor，我们就用它处理错误。

就像 `Maybe` 可以有个 `maybe` 一样，`Either` 也可以有一个 `either`。两者的用法类似，但 `either` 接受两个函数（而不是一个）和一个静态值为参数。这两个函数的返回值类型一致：

```
//  either :: (a -> c) -> (b -> c) -> Either a b -> c
var either = curry(function(f, g, e) {
  switch(e.constructor) {
    case Left: return f(e.__value);
    case Right: return g(e.__value);
  }
});

//  zoltar :: User -> _
var zoltar = compose(console.log, either(id, fortune), getAge(mo
ment()));

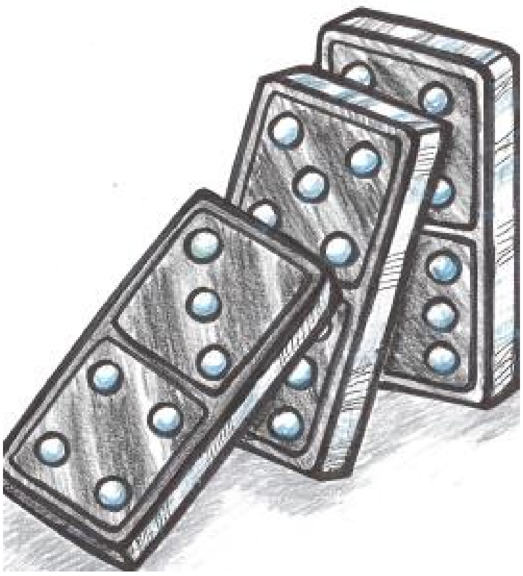
zoltar({birthdate: '2005-12-12'});
// "If you survive, you will be 10"
// undefined

zoltar({birthdate: 'balloons!'});
// "Birth date could not be parsed"
// undefined
```

终于用了一回那个神秘的 `id` 函数！其实它就是简单地复制了 `Left` 里的错误消息，然后把这个值传给 `console.log` 而已。通过强制在 `getAge` 内部进行错误处理，我们的算命程序更加健壮了。结果就是，要么告诉用户一个残酷的事实并像算命师那样跟他击掌，要么就继续运行程序。好了，现在我们已经准备好去学习一个完全不同类型的 functor 了。

王老先生有作用...

（译者注：原标题是“Old McDonald had Effects...”，源于美国儿歌“Old McDonald Had a Farm”。）



在关于纯函数的的那一章（即第 3 章）里，有一个很奇怪的例子。这个例子中的函数会产生副作用，但是我们通过把它包裹在另一个函数里的方式把它变得看起来像一个纯函数。这里还有一个类似的例子：

```
//  getFromStorage :: String -> (_ -> String)
var getFromStorage = function(key) {
  return function() {
    return localStorage[key];
  }
}
```

要是我们没把 `getFromStorage` 包在另一个函数里，它的输出值就是不定的，会随外部环境变化而变化。有了这个结实的包裹函数（wrapper），同一个输入就总能返回同一个输出：一个从 `localStorage` 里取出某个特定的元素的函数。就这样（也许再高唱几句赞美圣母的赞歌）我们洗涤了心灵，一切都得到了宽恕。

然而，这并没有多大的用处，你说是不是。就像是你收藏的全新未拆封的玩偶，不能拿出来玩有什么意思。所以要是能有办法进到这个容器里面，拿到它藏在那儿的东西就好了...办法是有的，请看 `IO` ：

```
var IO = function(f) {
  this.__value = f;
}

IO.of = function(x) {
  return new IO(function() {
    return x;
  });
}

IO.prototype.map = function(f) {
  return new IO(_.compose(f, this.__value));
}
```

`IO` 跟之前的 functor 不同的地方在于，它的 `__value` 总是一个函数。不过我们不把它当作一个函数——实现的细节我们最好先不管。这里发生的事情跟我们在 `getFromStorage` 那里看到的一模一样：`IO` 把非纯执行动作（impure action）捕获到包裹函数里，目的是延迟执行这个非纯动作。就这一点而言，我们认为 `IO` 包含的是被包裹的执行动作的返回值，而不是包裹函数本身。这在 `of` 函数里很明显：`IO(function(){ return x })` 仅仅是为了延迟执行，其实我们得到的是 `IO(x)`。

来用用看：

```
//  io_window_ :: IO Window
var io_window = new IO(function(){ return window; });

io_window.map(function(win){ return win.innerWidth });
// IO(1430)

io_window.map(_.prop('location')).map(_.prop('href')).map(split(
'/'));
// IO(["http:", "", "localhost:8000", "blog", "posts"])


//  $ :: String -> IO [DOM]
var $ = function(selector) {
  return new IO(function(){ return document.querySelectorAll(sel
ector); });
}

$('#myDiv').map(head).map(function(div){ return div.innerHTML; }
);
// IO('I am some inner html')
```

这里， `io_window` 是一个真正的 `IO` ，我们可以直接对它使用 `map` 。至于 `$` ，则是一个函数，调用后会返回一个 `IO` 。我把这里的返回值都写成了概念性的，这样就更加直观；不过实际的返回值是 `{ __value: [Function] }` 。当调用 `IO` 的 `map` 的时候，我们把传进来的函数放在了 `map` 函数里的组合的最末端（也就是最左边），反过来这个函数就成为了新的 `IO` 的新 `__value` ，并继续下去。传给 `map` 的函数并没有运行，我们只是把它们压到一个“运行栈”的最末端而已，一个函数紧挨着另一个函数，就像小心摆放的多米诺骨牌一样，让人不敢轻易推倒。这种情形很容易叫人联想起“四人帮”（译者注：《设计模式》一书作者）提出的命令模式（command pattern）或者队列（queue）。

花点时间找回你关于 functor 的直觉吧。把实现细节放在一边不管，你应该就能自然而然地对各种各样的容器使用 `map` 了，不管它是多么奇特怪异。这种伪超自然的力量要归功于 functor 的定律，我们将在本章末尾对此作一番探索。无论如何，我们终于可以在不牺牲代码纯粹性的情况下，随意使用这些不纯的值了。

好了，我们已经把野兽关进了笼子。但是，在某一时刻还是要把它放出来。因为对 `IO` 调用 `map` 已经积累了太多不纯的操作，最后再运行它无疑会打破平静。问题是在哪里，什么时候打开笼子的开关？而且有没有可能我们只运行 `IO` 却不让不纯的操作弄脏双手？答案是可以的，只要把责任推到调用者身上就行了。我们的纯代码，尽管阴险狡诈诡计多端，但是却始终保持一副清白无辜的模样，反而是实际运行 `IO` 并产生了作用的调用者，背了黑锅。来看一个具体的例子。

```
////// 纯代码库: lib/params.js ///////

//  url :: IO String
var url = new IO(function() { return window.location.href; });

//  toPairs =  String -> [[String]]
var toPairs = compose(map(split('=')), split('&'));

//  params :: String -> [[String]]
var params = compose(toPairs, last, split('?'));

//  findParam :: String -> IO Maybe [String]
var findParam = function(key) {
  return map(compose(Maybe.of, filter(compose(eq(key), head)), p
arams), url);
};

////// 非纯调用代码: main.js ///////

// 调用 __value() 来运行它！
findParam("searchTerm").__value();
// Maybe(['searchTerm', 'wafflehouse'])
```

lib/params.js 把 `url` 包裹在一个 `IO` 里，然后把这头野兽传给了调用者；一双手保持的非常干净。你可能也注意到了，我们把容器也“压栈”了，要知道创建一个 `IO(Maybe([x]))` 没有任何不合理的地方。我们这个“栈”有三层 functor（`Array` 是最有资格成为 mappable 的容器类型），令人印象深刻。

有件事困扰我很久了，现在我必须得说出来：`IO` 的 `__value` 并不是它包含的值，也不是像两个下划线暗示那样是一个私有属性。`__value` 是手榴弹的弹栓，只应该被调用者以最公开的方式拉动。为了提醒用户它的变化无常，我们把它重命名为 `unsafePerformIO` 看看。

```
var IO = function(f) {
  this.unsafePerformIO = f;
}

IO.prototype.map = function(f) {
  return new IO(_.compose(f, this.unsafePerformIO));
}
```

看，这就好多了。现在调用的代码就变成了 `findParam("searchTerm").unsafePerformIO()`，对应用程序的用户（以及本书读者）来说，这简直就直白得不能再直白了。

`IO` 会成为一个忠诚的伴侣，帮助我们驯化那些狂野的非纯操作。下一节我们将学习一种跟 `IO` 在精神上相似，但是用法上又千差万别的类型。

异步任务

回调（callback）是通往地狱的狭窄的螺旋阶梯。它们是埃舍尔（译者注：荷兰版画艺术家）设计的控制流。看到一个个嵌套的回调挤在大小括号搭成的架子上，让人不由自主地联想到地牢里的灵薄狱（还能再低点么！）（译者注：灵薄狱即 limbo，基督教中地狱边缘之意）。光是想到这样的回调就让我幽闭恐怖症发作了。不过别担心，处理异步代码，我们有一种更好的方式，它的名字以“F”开头。

这种方式的内部机制过于复杂，复杂得哪怕我唾沫横飞也很难讲清楚。所以我们就直接用 Quildreen Motta 的 Folktale 里的 `Data.Task` （之前是 `Data.Future`）。来见证一些例子吧：

```
// Node readfile example:
//========================

var fs = require('fs');

//  readFile :: String -> Task(Error, JSON)
var readFile = function(filename) {
  return new Task(function(reject, result) {
    fs.readFile(filename, 'utf-8', function(err, data) {
      err ? reject(err) : result(data);
    });
  });
};

readFile("metamorphosis").map(split('\n')).map(head);
// Task("One morning, as Gregor Samsa was waking up from anxious
 dreams, he discovered that
// in bed he had been changed into a monstrous verminous bug.")



// jQuery getJSON example:
//========================

//  getJSON :: String -> {} -> Task(Error, JSON)
var getJSON = curry(function(url, params) {
  return new Task(function(reject, result) {
    $.getJSON(url, params, result).fail(reject);
  });
});

getJSON('/video', {id: 10}).map(_.prop('title'));
// Task("Family Matters ep 15")

// 传入普通的实际值也没问题
Task.of(3).map(function(three){ return three + 1 });
// Task(4)
```

例子中的 `reject` 和 `result` 函数分别是失败和成功的回调。正如你看到的，我们只是简单地调用 `Task` 的 `map` 函数，就能操作将来的值，好像这个值就在那儿似的。到现在 `map` 对你来说应该不稀奇了。

如果熟悉 promise 的话，你该能认出来 `map` 就是 `then` ，`Task` 就是一个 promise。如果不熟悉你也不必气馁，反正我们也不会用它，因为它并不纯；但刚才的类比还是成立的。

与 `IO` 类似，`Task` 在我们给它绿灯之前是不会运行的。事实上，正因为它要等我们的命令，`IO` 实际就被纳入到了 `Task` 名下，代表所有的异步操作——`readFile` 和 `getJSON` 并不需要一个额外的 `IO` 容器来变纯。更重要的是，当我们调用它的 `map` 的时候，`Task` 工作的方式与 `IO` 几无差别：都是把对未来的操作的指示放在一个时间胶囊里，就像家务列表（chore chart）那样——真是一种精密的拖延术。

我们必须调用 `fork` 方法才能运行 `Task`，这种机制与 `unsafePerformIO` 类似。但也有不同，不同之处就像 `fork` 这个名称表明的那样，它会 fork 一个子进程运行它接收到的参数代码，其他部分的执行不受影响，主线程也不会阻塞。当然这种效果也可以用其他一些技术比如线程实现，但这里的这种方法工作起来就像是一个普通的异步调用，而且 event loop 能够不受影响地继续运转。我们来看一下 `fork`：

```
// Pure application
//=====================
// blogTemplate :: String

//  blogPage :: Posts -> HTML
var blogPage = Handlebars.compile(blogTemplate);

//  renderPage :: Posts -> HTML
var renderPage = compose(blogPage, sortBy('date'));

//  blog :: Params -> Task(Error, HTML)
var blog = compose(map(renderPage), getJSON('/posts'));


// Impure calling code
//=====================
blog({}).fork(
  function(error){ $("#error").html(error.message); },
  function(page){ $("#main").html(page); }
);

$('#spinner').show();
```

调用 `fork` 之后，`Task` 就赶紧跑去找一些文章，渲染到页面上。与此同时，我们在页面上展示一个 spinner，因为 `fork` 不会等收到响应了才执行它后面的代码。最后，我们要么把文章展示在页面上，要么就显示一个出错信息，视 `getJSON` 请求是否成功而定。

花点时间思考下这里的控制流为何是线性的。我们只需要从下读到上，从右读到左就能理解代码，即便这段程序实际上会在执行过程中到处跳来跳去。这种方式使得阅读和理解应用程序的代码比那种要在各种回调和错误处理代码块之间跳跃的方式容易得多。

天哪，你看到了么，`Task` 居然也包含了 `Either`！没办法，为了能处理将来可能出现的错误，它必须得这么做，因为普通的控制流在异步的世界里不适用。这自然是好事一桩，因为它天然地提供了充分的“纯”错误处理。

就算是有了 `Task` ，`IO` 和 `Either` 这两个 functor 也照样能派上用场。待我举个简单例子向你说明一种更复杂、更假想的情况，虽然如此，这个例子还是能够说明我的目的。

```
// Postgres.connect :: Url -> IO DbConnection
// runQuery :: DbConnection -> ResultSet
// readFile :: String -> Task Error String

// Pure application
//=====================

//  dbUrl :: Config -> Either Error Url
var dbUrl = function(c) {
  return (c.uname && c.pass && c.host && c.db)
    ? Right.of("db:pg://"+c.uname+":"+c.pass+"@"+c.host+"5432/"+c.db)
    : Left.of(Error("Invalid config!"));
}

//  connectDb :: Config -> Either Error (IO DbConnection)
var connectDb = compose(map(Postgres.connect), dbUrl);

//  getConfig :: Filename -> Task Error (Either Error (IO DbConnection))
var getConfig = compose(map(compose(connectDB, JSON.parse)), readFile);



// Impure calling code
//=====================
getConfig("db.json").fork(
  logErr("couldn't read file"), either(console.log, map(runQuery))
);
```

这个例子中，我们在 `readFile` 成功的那个代码分支里利用了 `Either` 和 `IO` 。`Task` 处理异步读取文件这一操作当中的不"纯"性，但是验证 config 的合法性以及连接数据库则分别使用了 `Either` 和 `IO` 。所以你看，我们依然在同步地跟所有事物打交道。

例子我还可以再举一些，但是就到此为止吧。这些概念就像 `map` 一样简单。

实际当中，你很有可能在一个工作流中跑好几个异步任务，但我们还没有完整学习容器的 api 来应对这种情况。不必担心，我们很快就会去学习 monad 之类的概念。不过，在那之前，我们得先检查下所有这些背后的数学知识。

一点理论

前面提到，functor 的概念来自于范畴学，并满足一些定律。我们先来探索这些实用的定律。

```
// identity
map(id) === id;

// composition
compose(map(f), map(g)) === map(compose(f, g));
```

同一律很简单，但是也很重要。因为这些定律都是可运行的代码，所以我们完全可以在我们自己的 functor 上试验它们，验证它们是否成立。

```
var idLaw1 = map(id);
var idLaw2 = id;

idLaw1(Container.of(2));
//=> Container(2)

idLaw2(Container.of(2));
//=> Container(2)
```

看到没，它们是相等的。接下来看一看组合。

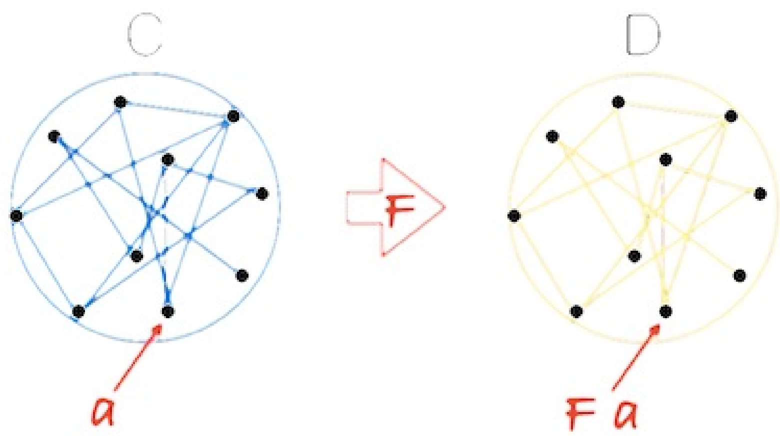
```
var compLaw1 = compose(map(concat(" world")), map(concat(" cruel")));
var compLaw2 = map(compose(concat(" world"), concat(" cruel")));

compLaw1(Container.of("Goodbye"));
//=> Container('Goodbye cruel world')

compLaw2(Container.of("Goodbye"));
//=> Container('Goodbye cruel world')
```

在范畴学中，functor 接受一个范畴的对象和态射（morphism），然后把它们映射（map）到另一个范畴里去。根据定义，这个新范畴一定会有一个单位元（identity），也一定能够组合态射；我们无须验证这一点，前面提到的定律保证这些东西会在映射后得到保留。

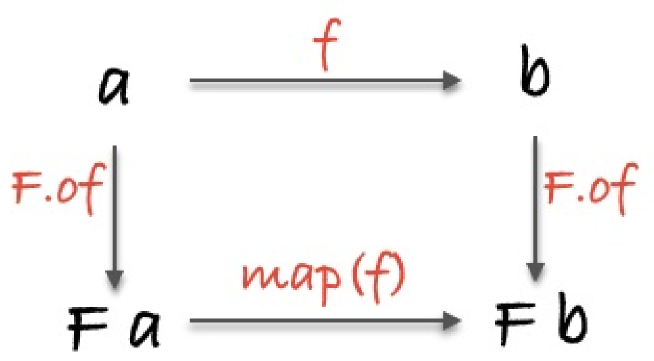
可能我们关于范畴的定义还是有点模糊。你可以把范畴想象成一个有着多个对象的网络，对象之间靠态射连接。那么 functor 可以把一个范畴映射到另外一个，而且不会破坏原有的网络。如果一个对象 `a` 属于源范畴 `C`，那么通过 functor `F` 把 `a` 映射到目标范畴 `D` 上之后，就可以使用 `F a` 来指代 `a` 对象（把这些字母拼起来是什么？！）。可能看图会更容易理解：



比如，`Maybe` 就把类型和函数的范畴映射到这样一个范畴：即每个对象都有可能不存在，每个态射都有空值检查的范畴。这个结果在代码中的实现方式是用 `map` 包裹每一个函数，用 functor 包裹每一个类型。这样就能保证每个普通的类型和函数都能在新环境下继续使用组合。从技术上讲，代码中的 functor 实际上是把范畴

映射到了一个包含类型和函数的子范畴（sub category）上，使得这些 functor 成为了一种新的特殊的 endofunctor。但出于本书的目的，我们认为它就是一个不同的范畴。

可以用一张图来表示这种态射及其对象的映射：



这张图除了能表示态射借助 functor `F` 完成从一个范畴到另一个范畴的映射之外，我们发现它还符合交换律，也就是说，顺着箭头的方向往前，形成的每一个路径都指向同一个结果。不同的路径意味着不同的行为，但最终都会得到同一个数据类型。这种形式化给了我们原则性的方式去思考代码——无须分析和评估每一个单独的场景，只管可以大胆地应用公式即可。来看一个具体的例子。

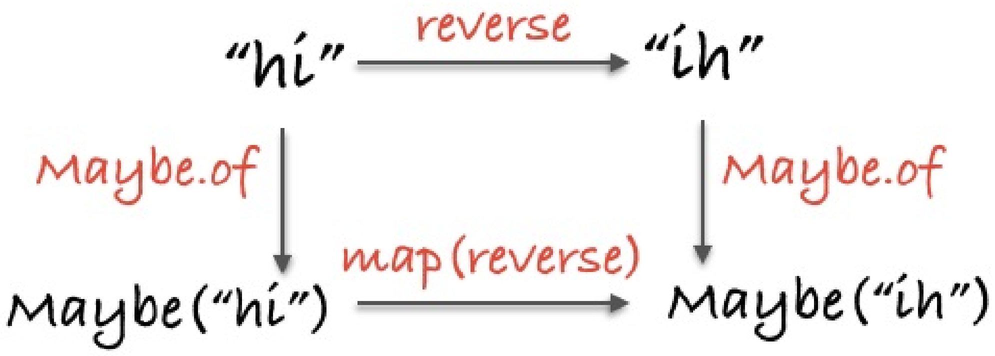
```
//  topRoute :: String -> Maybe(String)
var topRoute = compose(Maybe.of, reverse);

//  bottomRoute :: String -> Maybe(String)
var bottomRoute = compose(map(reverse), Maybe.of);


topRoute("hi");
// Maybe("ih")

bottomRoute("hi");
// Maybe("ih")
```

或者看图：



根据所有 functor 都有的特性，我们可以立即理解代码，重构代码。

functor 也能嵌套使用：

```
var nested = Task.of([Right.of("pillows"), Left.of("no sleep for
 you")]);

map(map(map(toUpperCase)), nested);
// Task([Right("PILLOWS"), Left("no sleep for you")])
```

`nested` 是一个将来的数组，数组的元素有可能是程序抛出的错误。我们使用 `map` 剥开每一层的嵌套，然后对数组的元素调用传递进去的函数。可以看到，这中间没有回调、`if/else` 语句和 `for` 循环，只有一个明确的上下文。的确，我们必须要 `map(map(map(f)))` 才能最终运行函数。不想这么做的话，可以组合 functor。是的，你没听错：

```
var Compose = function(f_g_x){
  this.getCompose = f_g_x;
}

Compose.prototype.map = function(f){
  return new Compose(map(map(f), this.getCompose));
}

var tmd = Task.of(Maybe.of("Rock over London"))

var ctmd = new Compose(tmd);

map(concat(", rock on, Chicago"), ctmd);
// Compose(Task(Maybe("Rock over London, rock on, Chicago")))

ctmd.getCompose;
// Task(Maybe("Rock over London, rock on, Chicago"))
```

看，只有一个 `map`。functor 组合是符合结合律的，而且之前我们定义的 `Container` 实际上是一个叫 `Identity` 的 functor。identity 和可结合的组合也能产生一个范畴，这个特殊的范畴的对象是其他范畴，态射是 functor。这实在太伤脑筋了，所以我们不会深入这个问题，但是赞叹一下这种模式的结构性含义，或者它的简单的抽象之美也是好的。

总结

我们已经认识了几个不同的 functor，但它们的数量其实是无限的。有一些值得注意的可迭代数据类型（iterable data structure）我们没有介绍，像 tree、list、map 和 pair 等，以及所有你能说出来的。eventstream 和 observable 也都是 functor。其他的 functor 可能就是拿来做封装或者仅仅是模拟类型。我们身边到处都有 functor 的身影，本书也将会大量使用它们。

用多个 functor 参数调用一个函数怎么样呢？处理一个由不纯的或者异步的操作组成的有序序列怎么样呢？要应对这个什么都装在盒子里的世界，目前我们工具箱里的工具还不全。下一章，我们将直奔 monad 而去。

第 9 章: Monad

练习

```
require('../../support');
var Task = require('data.task');
var _ = require('ramda');

// 练习 1
// ==========
// 使用 _.add(x,y) 和 _.map(f,x) 创建一个能让 functor 里的值增加的函数

var ex1 = undefined



//练习 2
// ==========
// 使用 _.head 获取列表的第一个元素
var xs = Identity.of(['do', 'ray', 'me', 'fa', 'so', 'la', 'ti',
'do']);

var ex2 = undefined



// 练习 3
// ==========
// 使用 safeProp 和 _.head 找到 user 的名字的首字母
var safeProp = _.curry(function (x, o) { return Maybe.of(o[x]);
});

var user = { id: 2, name: "Albert" };

var ex3 = undefined


// 练习 4
// ==========
// 使用 Maybe 重写 ex4，不要有 if 语句

```

```
var ex4 = function (n) {
  if (n) { return parseInt(n); }
};

var ex4 = undefined




// 练习 5
// ==========
// 写一个函数，先 getPost 获取一篇文章，然后 toUpperCase 让这片文章标题
变为大写

// getPost :: Int -> Future({id: Int, title: String})
var getPost = function (i) {
  return new Task(function(rej, res) {
    setTimeout(function(){
      res({id: i, title: 'Love them futures'})
    }, 300)
  });
}

var ex5 = undefined




// 练习 6
// ==========
// 写一个函数，使用 checkActive() 和 showWelcome() 分别允许访问或返回
错误

var showWelcome = _.compose(_.add( "Welcome "), _.prop('name'))

var checkActive = function(user) {
 return user.active ? Right.of(user) : Left.of('Your account is
not active')
}

var ex6 = undefined
```

```
// 练习 7
// ==========
// 写一个验证函数，检查参数是否 length > 3。如果是就返回 Right(x)，否则就返回
// Left("You need > 3")

var ex7 = function(x) {
  return undefined // <--- write me. (don't be pointfree)
}



// 练习 8
// ==========
// 使用练习 7 的 ex7 和 Either 构造一个 functor，如果一个 user 合法就保存它，否则
// 返回错误消息。别忘了 either 的两个参数必须返回同一类型的数据。

var save = function(x){
  return new IO(function(){
    console.log("SAVED USER!");
    return x + '-saved';
  });
}

var ex8 = undefined
```

Monad

pointed functor

在继续后面的内容之前，我得向你坦白一件事：关于我们先前创建的容器类型上的 `of` 方法，我并没有说出它的全部实情。真实情况是，`of` 方法不是用来避免使用 `new` 关键字的，而是用来把值放到默认最小化上下文（default minimal context）中的。是的，`of` 没有真正地取代构造器——它是一个我们称之为 *pointed* 的重要接口的一部分。

> *pointed functor* 是实现了 `of` 方法的 functor。

这里的关键是把任意值丢到容器里然后开始到处使用 `map` 的能力。

```
IO.of("tetris").map(concat(" master"));
// IO("tetris master")

Maybe.of(1336).map(add(1));
// Maybe(1337)

Task.of([{id: 2}, {id: 3}]).map(_.prop('id'));
// Task([2,3])

Either.of("The past, present and future walk into a bar...").map
(
  concat("it was tense.")
);
// Right("The past, present and future walk into a bar...it was
tense.")
```

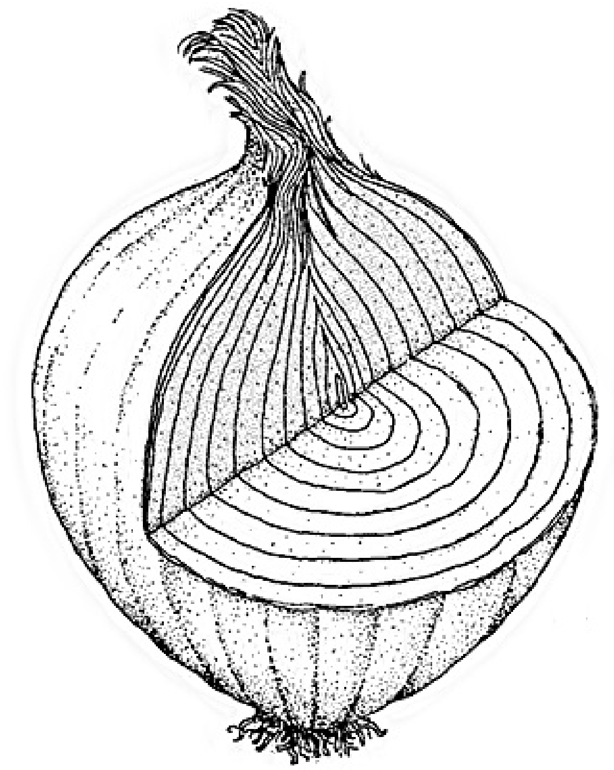
如果你还记得，`IO` 和 `Task` 的构造器接受一个函数作为参数，而 `Maybe` 和 `Either` 的构造器可以接受任意值。实现这种接口的动机是，我们希望能有一种通用、一致的方式往 functor 里填值，而且中间不会涉及到复杂性，也不会涉及到对构造器的特定要求。“默认最小化上下文”这个术语可能不够精确，但是却很好地传达了这种理念：我们希望容器类型里的任意值都能发生 `lift`，然后像所有的 functor 那样再 `map` 出去。

有件很重要的事我必须得在这里纠正，那就是， `Left.of` 没有任何道理可言，包括它的双关语也是。每个 functor 都要有一种把值放进去的方式，对 `Either` 来说，它的方式就是 `new Right(x)` 。我们为 `Right` 定义 `of` 的原因是，如果一个类型容器可以 `map` ，那它就应该 `map` 。看上面的例子，你应该会对 `of` 通常的工作模式有一个直观的印象，而 `Left` 破坏了这种模式。

你可能已经听说过 `pure` 、 `point` 、 `unit` 和 `return` 之类的函数了，它们都是 `of` 这个史上最神秘函数的不同名称（译者注：此处原文是“international function of mystery”，源自恶搞《007》的电影 *Austin Powers: International Man of Mystery*，中译名《王牌大贱谍》）。 `of` 将在我们开始使用 monad 的时候显示其重要性，因为后面你会看到，手动把值放回容器是我们自己的责任。

要避免 `new` 关键字，可以借助一些标准的 JavaScript 技巧或者类库达到目的。所以从这里开始，我们就利用这些技巧或类库，像一个负责任的成年人那样使用 `of` 。我推荐使用 `folktale` 、 `ramda` 或 `fantasy-land` 里的 functor 实例，因为它们同时提供了正确的 `of` 方法和不依赖 `new` 的构造器。

混合比喻



你看，除了太空墨西哥卷（如果你听说过这个传言的话）（译者注：此处的传言似乎是说一个叫 Chris Hadfield 的宇航员在国际空间站做墨西哥卷的事，视频链接），monad 还被喻为洋葱。让我以一个常见的场景来说明这点：

```
// Support
// ===========================
var fs = require('fs');

//  readFile :: String -> IO String
var readFile = function(filename) {
  return new IO(function() {
    return fs.readFileSync(filename, 'utf-8');
  });
};

//  print :: String -> IO String
var print = function(x) {
  return new IO(function() {
    console.log(x);
    return x;
  });
}

// Example
// ===========================
//  cat :: IO (IO String)
var cat = compose(map(print), readFile);

cat(".git/config")
// IO(IO("[core]\nrepositoryformatversion = 0\n"))
```

这里我们得到的是一个 `IO` ，只不过它陷进了另一个 `IO` 。要想使用它，我们必须这样调用： `map(map(f))` ；要想观察它的作用，必须这样： `unsafePerformIO().unsafePerformIO()` 。

```
//  cat :: String -> IO (IO String)
var cat = compose(map(print), readFile);

//  catFirstChar :: String -> IO (IO String)
var catFirstChar = compose(map(map(head)), cat);

catFirstChar(".git/config")
// IO(IO("["))
```

尽管在应用中把这两个作用打包在一起没什么不好的，但总感觉像是在穿着两套防护服工作，结果就形成一个稀奇古怪的 API。再来看另一种情况：

```
//  safeProp :: Key -> {Key: a} -> Maybe a
var safeProp = curry(function(x, obj) {
  return new Maybe(obj[x]);
});

//  safeHead :: [a] -> Maybe a
var safeHead = safeProp(0);

//  firstAddressStreet :: User -> Maybe (Maybe (Maybe Street) )
var firstAddressStreet = compose(
  map(map(safeProp('street'))), map(safeHead), safeProp('address
es')
);

firstAddressStreet(
  {addresses: [{street: {name: 'Mulburry', number: 8402}, postco
de: "WC2N" }]}
);
// Maybe(Maybe(Maybe({name: 'Mulburry', number: 8402})))
```

这里的 functor 同样是嵌套的，函数中三个可能的失败都用了 `Maybe` 做预防也很干净整洁，但是要让最后的调用者调用三次 `map` 才能取到值未免也太无礼了点——我们和它才刚刚见面而已。这种嵌套 functor 的模式会时不时地出现，而且是 monad 的主要使用场景。

我说过 monad 像洋葱，那是因为当我们用 `map` 剥开嵌套的 functor 以获取它里面的值的时候，就像剥洋葱一样让人忍不住想哭。不过，我们可以擦干眼泪，做个深呼吸，然后使用一个叫作 `join` 的方法。

```
var mmo = Maybe.of(Maybe.of("nunchucks"));
// Maybe(Maybe("nunchucks"))

mmo.join();
// Maybe("nunchucks")

var ioio = IO.of(IO.of("pizza"));
// IO(IO("pizza"))

ioio.join()
// IO("pizza")

var ttt = Task.of(Task.of(Task.of("sewers")));
// Task(Task(Task("sewers")));

ttt.join()
// Task(Task("sewers"))
```

如果有两层相同类型的嵌套，那么就可以用 `join` 把它们压扁到一块去。这种结合的能力，functor 之间的联姻，就是 monad 之所以成为 monad 的原因。来看看它更精确的完整定义：

> monad 是可以变扁（flatten）的 pointed functor。

一个 functor，只要它定义个了一个 `join` 方法和一个 `of` 方法，并遵守一些定律，那么它就是一个 monad。`join` 的实现并不太复杂，我们来为 `Maybe` 定义一个：

```
Maybe.prototype.join = function() {
  return this.isNothing() ? Maybe.of(null) : this.__value;
}
```

看，就像子宫里双胞胎中的一个吃掉另一个那么简单。如果有一个 `Maybe(Maybe(x))`，那么 `.__value` 将会移除多余的一层，然后我们就能安心地从那开始进行 `map`。要不然，我们就将会只有一个 `Maybe`，因为从一开始就没有任何东西被 `map` 调用。

既然已经有了 `join` 方法，我们把 monad 魔法作用到 `firstAddressStreet` 例子上，看看它的实际作用：

```
//  join :: Monad m => m (m a) -> m a
var join = function(mma){ return mma.join(); }

//  firstAddressStreet :: User -> Maybe Street
var firstAddressStreet = compose(
  join, map(safeProp('street')), join, map(safeHead), safeProp('
addresses')
);

firstAddressStreet(
  {addresses: [{street: {name: 'Mulburry', number: 8402}, postco
de: "WC2N" }]}
);
// Maybe({name: 'Mulburry', number: 8402})
```

只要遇到嵌套的 `Maybe`，就加一个 `join`，防止它们从手中溜走。我们对 `IO` 也这么做试试看，感受下这种感觉。

```
IO.prototype.join = function() {
  return this.unsafePerformIO();
}
```

同样是简单地移除了一层容器。注意，我们还没有提及纯粹性的问题，仅仅是移除过度紧缩的包裹中的一层而已。

```
//  log :: a -> IO a
var log = function(x) {
  return new IO(function() { console.log(x); return x; });
}

//  setStyle :: Selector -> CSSProps -> IO DOM
var setStyle = curry(function(sel, props) {
  return new IO(function() { return jQuery(sel).css(props); });
});

//  getItem :: String -> IO String
var getItem = function(key) {
  return new IO(function() { return localStorage.getItem(key); }
);
};

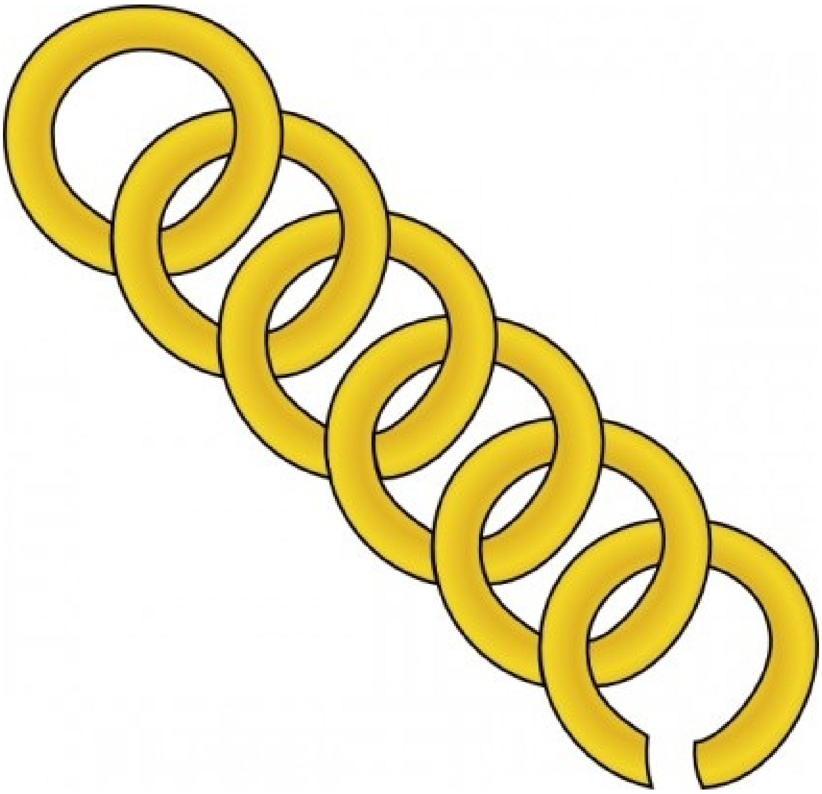
//  applyPreferences :: String -> IO DOM
var applyPreferences = compose(
  join, map(setStyle('#main')), join, map(log), map(JSON.parse),
 getItem
);


applyPreferences('preferences').unsafePerformIO();
// Object {backgroundColor: "green"}
// <div style="background-color: 'green'"/>
```

`getItem` 返回了一个 `IO String`，所以可以直接用 `map` 来解析它。`log` 和 `setStyle` 返回的都是 `IO`，所以必须要使用 `join` 来保证这里边的嵌套处于控制之中。

chain 函数

（译者注：此处标题原文是“My chain hits my chest”，是英国歌手 M.I.A 单曲 *Bad Girls* 的一句歌词。据说这首歌有体现女权主义。）



你可能已经从上面的例子中注意到这种模式了：我们总是在紧跟着 `map` 的后面调用 `join` 。让我们把这个行为抽象到一个叫做 `chain` 的函数里。

```
//  chain :: Monad m => (a -> m b) -> m a -> m b
var chain = curry(function(f, m){
  return m.map(f).join(); // 或者 compose(join, map(f))(m)
});
```

这里仅仅是把 map/join 套餐打包到一个单独的函数中。如果你之前了解过 monad，那你可能已经看出来 `chain` 叫做 `>>=` （读作 bind）或者 `flatMap` ；都是同一个概念的不同名称罢了。我个人认为 `flatMap` 是最准确的名称，但本书还是坚持使用 `chain` ，因为它是 JS 里接受程度最高的一个。我们用 `chain` 重构下上面两个例子：

```
// map/join
var firstAddressStreet = compose(
  join, map(safeProp('street')), join, map(safeHead), safeProp('
addresses')
);

// chain
var firstAddressStreet = compose(
  chain(safeProp('street')), chain(safeHead), safeProp('addresse
s')
);



// map/join
var applyPreferences = compose(
  join, map(setStyle('#main')), join, map(log), map(JSON.parse),
 getItem
);

// chain
var applyPreferences = compose(
  chain(setStyle('#main')), chain(log), map(JSON.parse), getItem
);
```

我把所有的 `map/join` 都替换为了 `chain`，这样代码就显得整洁了些。整洁固然是好事，但 `chain` 的能力却不止于此——它更多的是龙卷风而不是吸尘器。因为 `chain` 可以轻松地嵌套多个作用，因此我们就能以一种纯函数式的方式来表示序列（sequence）和 变量赋值（variable assignment）。

```
// getJSON :: Url -> Params -> Task JSON
// querySelector :: Selector -> IO DOM


getJSON('/authenticate', {username: 'stale', password: 'crackers'
})
  .chain(function(user) {
    return getJSON('/friends', {user_id: user.id});
});
// Task([{name: 'Seimith', id: 14}, {name: 'Ric', id: 39}]);


querySelector("input.username").chain(function(uname) {
  return querySelector("input.email").chain(function(email) {
    return IO.of(
      "Welcome " + uname.value + " " + "prepare for spam at " +
email.value
    );
  });
});
// IO("Welcome Olivia prepare for spam at olivia@tremorcontrol.n
et");


Maybe.of(3).chain(function(three) {
  return Maybe.of(2).map(add(three));
});
// Maybe(5);


Maybe.of(null).chain(safeProp('address')).chain(safeProp('street'
));
// Maybe(null);
```

本来我们可以用 `compose` 写上面的例子，但这将需要几个帮助函数，而且这种风格怎么说都要通过闭包进行明确的变量赋值。相反，我们使用了插入式的 `chain` 。顺便说一下， `chain` 可以自动从任意类型的 `map` 和 `join` 衍生出来，就像这样： `t.prototype.chain = function(f) { return`

`this.map(f).join(); }` 。如果手动定义 `chain` 能让你觉得性能会好点的话（实际上并不会），我们也可以手动定义它，尽管还必须要费力保证函数功能的正确性——也就是说，它必须与紧接着后面有 `join` 的 `map` 相等。如果 `chain` 是简单地通过结束调用 `of` 后把值放回容器这种方式定义的，那么就会造成一个有趣的后果，即可以从 `chain` 那里衍生出一个 `map` 。同样地，我们还可以用 `chain(id)` 定义 `join` 。听起来好像是在跟魔术师玩德州扑克，魔术师想要什么牌就有什么牌；但是就像大部分的数学理论一样，所有这些原则性的结构都是相互关联的。fantasyland 仓库中提到了许多上述衍生概念，这个仓库也是 JavaScript 官方的代数数据结构（algebraic data types）标准。

好了，我们来看上面的例子。第一个例子中，可以看到两个 `Task` 通过 `chain` 连接形成了一个异步操作的序列——它先获取 `user` ，然后用 `user.id` 查找 `user` 的 `friends` 。 `chain` 避免了 `Task(Task([Friend]))` 这种情况。

第二个例子是用 `querySelector` 查找几个 input 然后创建一条欢迎信息。注意看我们是如何在最内层的函数里访问 `uname` 和 `email` 的——这是函数式变量赋值的绝佳表现。因为 `IO` 大方地把它的值借给了我们，我们也要负起以同样方式把值放回去的责任——不能辜负它的信任（还有整个程序的信任）。 `IO.of` 非常适合做这件事，同时它也解释了为何 pointed 这一特性是 monad 接口得以存在的重要前提。不过， `map` 也能返回正确的类型：

```
querySelector("input.username").chain(function(uname) {
  return querySelector("input.email").map(function(email) {
    return "Welcome " + uname.value + " prepare for spam at " +
email.value;
  });
});
// IO("Welcome Olivia prepare for spam at olivia@tremorcontrol.n
et");
```

最后两个例子用了 `Maybe` 。因为 `chain` 其实是在底层调用了 `map` ，所以如果遇到 `null` ，代码就会立刻停止运行。

如果觉得这些例子不太容易理解，你也不必担心。多跑跑代码，多琢磨琢磨，把代码拆开来研究研究，再把它们拼起来看看。总之记住，返回的如果是“普通”值就用 `map` ，如果是 `functor` 就用 `chain` 。

这里我得提醒一下，上述方式对两个不同类型的嵌套容器是不适用的。functor 组合，以及后面会讲到的 monad transformer 可以帮助我们应对这种情况。

炫耀

这种容器编程风格有时也能造成困惑，我们不得不努力理解一个值到底嵌套了几层容器，或者需要用 `map` 还是 `chain`（很快我们就会认识更多的容器类型）。使用一些技巧，比如重写 `inspect` 方法之类，能够大幅提高 debug 的效率。后面我们也会学习如何创建一个“栈”，使之能够处理任何丢给它的作用（effects）。不过，有时候也需要权衡一下是否值得这样做。

我很乐意挥起 monad 之剑，向你展示这种编程风格的力量。就以读一个文件，然后就把它直接上传为例吧：

```
// readFile :: Filename -> Either String (Future Error String)
// httpPost :: String -> Future Error JSON

//  upload :: String -> Either String (Future Error JSON)
var upload = compose(map(chain(httpPost('/uploads'))), readFile)
;
```

这里，代码不止一次在不同的分支执行。从类型签名可以看出，我们预防了三个错误—— `readFile` 使用 `Either` 来验证输入（或许还有确保文件名存在）；`readFile` 在读取文件的时候可能会出错，错误通过 `readFile` 的 `Future` 表示；文件上传可能会因为各种各样的原因出错，错误通过 `httpPost` 的 `Future` 表示。我们就这么随意地使用 `chain` 实现了两个嵌套的、有序的异步执行动作。

所有这些操作都是在一个从左到右的线性流中完成的，是完完全全纯的、声明式的代码，是可以等式推导（equational reasoning）并拥有可靠特性（reliable properties）的代码。我们没有被迫使用不必要甚至令人困惑的变量名，我们的 `upload` 函数符合通用接口而不是特定的一次性接口。这些都是在一行代码中完成的啊！

让我们来跟标准的命令式的实现对比一下：

```js
//  upload :: String -> (String -> a) -> Void
var upload = function(filename, callback) {
  if(!filename) {
    throw "You need a filename!";
  } else {
    readFile(filename, function(err, contents) {
      if(err) throw err;
      httpPost(contents, function(err, json) {
        if(err) throw err;
        callback(json);
      });
    });
  }
}
```

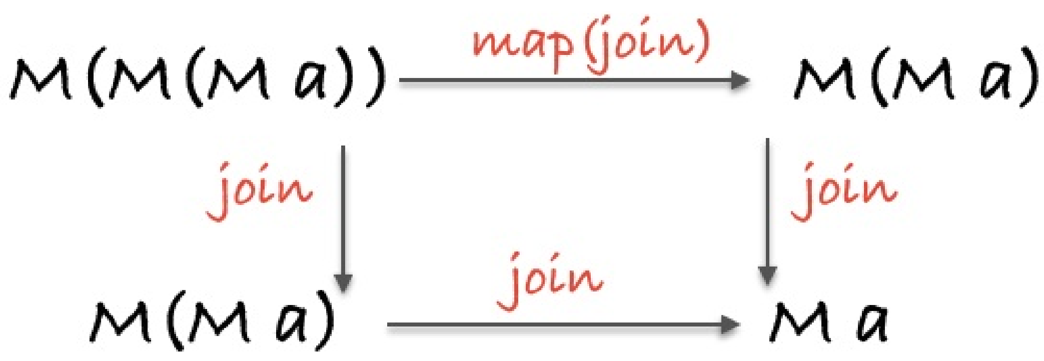
看看，这简直就是魔鬼的算术（译者注：此处原文是“the devil's arithmetic”，为美国 1988 年出版的历史小说，讲述一个犹太小女孩穿越到 1942 年的集中营的故事。此书亦有同名改编电影，中译名《穿梭集中营》），我们就像一颗弹珠一样在变幻莫测的迷宫中穿梭。无法想象如果这是一个典型的应用，而且一直在改变变量会怎样——我们肯定会像陷入沥青坑那样无所适从。

理论

我们要看的第一条定律是结合律，但可能不是你熟悉的那个结合律。

```js
// 结合律
compose(join, map(join)) == compose(join, join)
```

这些定律表明了 monad 的嵌套本质，所以结合律关心的是如何让内层或外层的容器类型 `join`，然后取得同样的结果。用一张图来表示可能效果会更好：

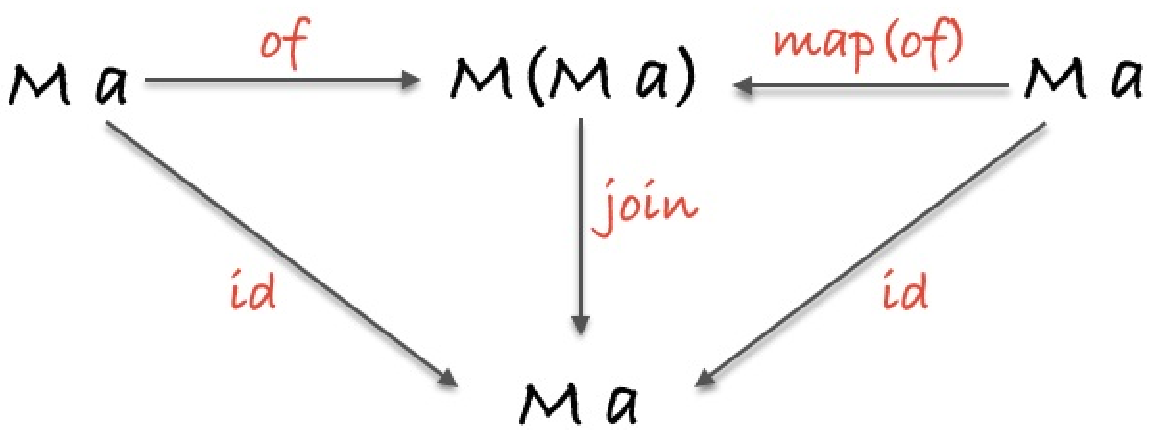


从左上角往下，先用 `join` 合并 `M(M(M a))` 最外层的两个 `M` ，然后往右，再调用一次 `join` ，就得到了我们想要的 `M a` 。或者，从左上角往右，先打开最外层的 `M` ，用 `map(join)` 合并内层的两个 `M` ，然后再向下调用一次 `join` ，也能得到 `M a` 。不管是先合并内层还是先合并外层的 `M` ，最后都会得到相同的 `M a` ，所以这就是结合律。值得注意的一点是 `map(join) != join` 。两种方式的中间步骤可能会有不同的值，但最后一个 `join` 调用后最终结果是一样的。

第二个定律与结合律类似：

```
// 同一律 (M a)
compose(join, of) == compose(join, map(of)) == id
```

这表明，对任意的 monad `M` ， `of` 和 `join` 相当于 `id` 。也可以使用 `map(of)` 由内而外实现相同效果。我们把这个定律叫做“三角同一律”（triangle identity），因为把它图形化之后就像一个三角形：



如果从左上角开始往右，可以看到 `of` 的确把 `M a` 丢到另一个 `M` 容器里去了。然后再往下 `join`，就得到了 `M a`，跟一开始就调用 `id` 的结果一样。从右上角往左，可以看到如果我们通过 `map` 进到了 `M` 里面，然后对普通值 `a` 调用 `of`，最后得到的还是 `M (M a)`；再调用一次 `join` 将会把我们带回原点，即 `M a`。

我要说明一点，尽管这里我写的是 `of`，实际上对任意的 monad 而言，都必须要使用明确的 `M.of`。

我已经见过这些定律了，同一律和结合律，以前就在哪儿见过...等一下，让我想想...是的！它们是范畴遵循的定律！不过这意味着我们需要一个组合函数来给出一个完整定义。见证吧：

```
var mcompose = function(f, g) {
  return compose(chain(f), chain(g));
}

// 左同一律
mcompose(M, f) == f

// 右同一律
mcompose(f, M) == f

// 结合律
mcompose(mcompose(f, g), h) == mcompose(f, mcompose(g, h))
```

毕竟它们是范畴学里的定律。monad 来自于一个叫 “Kleisli 范畴”的范畴，这个范畴里边所有的对象都是 monad，所有的态射都是联结函数（chained funtions）。我不是要在没有提供太多解释的情况下，拿范畴学里各式各样的概念来取笑你。我的目的是涉及足够多的表面知识，向你说明这中间的相关性，让你在关注日常实用特性之余，激发起对这些定律的兴趣。

总结

monad 让我们深入到嵌套的运算当中，使我们能够在完全避免回调金字塔（pyramid of doom）情况下，为变量赋值，运行有序的作用，执行异步任务等等。当一个值被困在几层相同类型的容器中时，monad 能够拯救它。借助

“pointed” 这个可靠的帮手，monad 能够借给我们从盒子中取出的值，而且知道我们会在结束使用后还给它。

是的，monad 非常强大，但我们还需要一些额外的容器函数。比如，假设我们想同时运行一个列表里的 api 调用，然后再搜集返回的结果，怎么办？是可以使用 monad 实现这个任务，但必须要等每一个 api 完成后才能调用下一个。合并多个合法性验证呢？我们想要的肯定是持续验证以搜集错误列表，但是 monad 会在第一个 `Left` 登场的时候停掉整个演出。

下一章，我们将看到 applicative functor 如何融入这个容器世界，以及为何在很多情况下它比 monad 更好用。

第 10 章: Applicative Functor

练习

```
// 练习 1
// ==========
// 给定一个 user，使用 safeProp 和 map/join 或 chain 安全地获取 sreet 的 name

var safeProp = _.curry(function (x, o) { return Maybe.of(o[x]);
});
var user = {
  id: 2,
  name: "albert",
  address: {
    street: {
      number: 22,
      name: 'Walnut St'
    }
  }
};

var ex1 = undefined;


// 练习 2
// ==========
```

```
// 使用 getFile 获取文件名并删除目录，所以返回值仅仅是文件，然后以纯的方式
打印文件

var getFile = function() {
  return new IO(function(){ return __filename; });
}

var pureLog = function(x) {
  return new IO(function(){
    console.log(x);
    return 'logged ' + x;
  });
}

var ex2 = undefined;



// 练习 3
// ==========
// 使用 getPost() 然后以 post 的 id 调用 getComments()
var getPost = function(i) {
  return new Task(function (rej, res) {
    setTimeout(function () {
      res({ id: i, title: 'Love them tasks' });
    }, 300);
  });
}

var getComments = function(i) {
  return new Task(function (rej, res) {
    setTimeout(function () {
      res([
        {post_id: i, body: "This book should be illegal"},
        {post_id: i, body: "Monads are like smelly shallots"}
      ]);
    }, 300);
  });
}
```

```
var ex3 = undefined;


// 练习 4
// ==========
// 用 validateEmail、addToMailingList 和 emailBlast 实现 ex4 的类型
签名

//  addToMailingList :: Email -> IO([Email])
var addToMailingList = (function(list){
  return function(email) {
    return new IO(function(){
      list.push(email);
      return list;
    });
  }
})([]);

function emailBlast(list) {
  return new IO(function(){
    return 'emailed: ' + list.join(',');
  });
}

var validateEmail = function(x){
  return x.match(/\S+@\S+\.\S+/) ? (new Right(x)) : (new Left('i
nvalid email'));
}

//  ex4 :: Email -> Either String (IO String)
var ex4 = undefined;
```

Applicative Functor

应用 applicative functor

考虑到其函数式的出身，**applicative functor** 这个名称堪称简单明了。函数式程序员最为人诟病的一点就是，总喜欢搞一些稀奇古怪的命名，比如 `mappend` 或者 `liftA4`。诚然，此类名称出现在数学实验室是再自然不过的，但是放在其他任何语境下，这些概念就都像是扮作达斯维达去汽车餐馆搞怪的人。（译者注：此处需要做些解释，1. 汽车餐馆（drive-thru）指的是那种不需要顾客下车就能提供服务的地方，比如麦当劳、星巴克等就会有这种 drive-thru；2. 达斯维达（Darth Vader）是《星球大战》系列主要反派角色，在美国大众文化中的有着广泛的影响力，其造型是很多人致敬模仿的对象；3. 由于 2 的缘故，美国一些星战迷会扮作 Darth Vader 去 drive-thru 点单，YouTube 上有不少这种搞怪视频；4. 作者使用这个“典故”是为了说明函数式里很多概念的名称有些“故弄玄虚”，而 applicative functor 是少数比较“正常”的。）

无论如何，applicative 这个名字应该能够向我们表明一些事实，告诉我们作为一个接口，它能为我们带来什么：那就是让不同 functor 可以相互应用（apply）的能力。

然而，你可能会问了，为何一个正常的、理性的人，比如你自己，会做这种“让不同 functor 相互应用”的事？而且，“相互应用”到底是什么意思？

要回答这些问题，我们可以从下面这个场景讲起，可能你已经碰到过这种场景了。假设有两个同类型的 functor，我们想把这两者作为一个函数的两个参数传递过去来调用这个函数。简单的例子比如让两个 `Container` 的值相加：

```
// 这样是行不通的，因为 2 和 3 都藏在瓶子里。
add(Container.of(2), Container.of(3));
//NaN

// 使用可靠的 map 函数试试
var container_of_add_2 = map(add, Container.of(2));
// Container(add(2))
```

这时候我们创建了一个 `Container`，它内部的值是一个局部调用的（partially applied）的函数。确切点讲就是，我们想让 `Container(add(2))` 中的 `add(2)` 应用到 `Container(3)` 中的 `3` 上来完成调用。也就是说，我们想把一个 functor 应用到另一个上。

巧的是，完成这种任务的工具已经存在了，即 `chain` 函数。我们可以先 `chain` 然后再 `map` 那个局部调用的 `add(2)`，就像这样：

```
Container.of(2).chain(function(two) {
  return Container.of(3).map(add(two));
});
```

只不过，这种方式有一个问题，那就是 monad 的顺序执行问题：所有的代码都只会在前一个 monad 执行完毕之后才执行。想想看，我们的这两个值足够强健且相互独立，如果仅仅为了满足 monad 的顺序要求而延迟 `Container(3)` 的创建，我觉得是非常没有必要的。

事实上，当遇到这种问题的时候，要是能够无需借助这些不必要的函数和变量，以一种简明扼要的方式把一个 functor 的值应用到另一个上去就好了。

瓶中之船



`ap` 就是这样一种函数，能够把一个 functor 的函数值应用到另一个 functor 的值上。把这句话快速地说上 5 遍。

```
Container.of(add(2)).ap(Container.of(3));
// Container(5)

// all together now
Container.of(2).map(add).ap(Container.of(3));
// Container(5)
```

这样就大功告成了，而且代码干净整洁。可以看到， `Container(3)` 从嵌套的 monad 函数的牢笼中释放了出来。需要再次强调的是，本例中的 `add` 是被 `map` 所局部调用（partially apply）的，所以 `add` 必须是一个 curry 函数。

可以这样定义一个 `ap` 函数：

```
Container.prototype.ap = function(other_container) {
  return other_container.map(this.__value);
}
```

记住， `this.__value` 是一个函数，将会接收另一个 functor 作为参数，所以我们只需 `map` 它。由此我们可以得出 applicative functor 的定义：

> applicative functor 是实现了 `ap` 方法的 pointed functor

注意 `pointed` 这个前提，这是非常重要的一个前提，下面的例子会说明这一点。

讲到这里，我已经感受到你的疑虑了（也或者是困惑和恐惧）；心态开放点嘛， `ap` 还是很有用的。在深入理解这个概念之前，我们先来探索一个特性。

```
F.of(x).map(f) == F.of(f).ap(F.of(x))
```

这行代码翻译成人类语言就是，map 一个 `f` 等价于 `ap` 一个值为 `f` 的 functor。或者更好的译法是，你既可以把 `x` 放到容器里然后调用 `map(f)` ，也可以同时让 `f` 和 `x` 发生 lift（参看第 8 章），然后对他们调用 `ap` 。这让我们能够以一种从左到右的方式编写代码：

```
Maybe.of(add).ap(Maybe.of(2)).ap(Maybe.of(3));
// Maybe(5)

Task.of(add).ap(Task.of(2)).ap(Task.of(3));
// Task(5)
```

细心的读者可能发现了，上述代码中隐约有普通函数调用的影子。没关系，我们稍后会学习 `ap` 的 pointfree 版本；暂时先把这当作此类代码的推荐写法。通过使用 `of`，每一个值都被输送到了各个容器里的奇幻之地，就像是在另一个平行世界里，每个程序都可以是异步的或者是 null 或者随便什么值，而且不管是什么，`ap` 都能在这个平行世界里针对这些值应用各种各样的函数。这就像是在一个瓶子中造船。

你注意到没？上例中我们使用了 `Task`，这是 applicative functor 主要的用武之地。现在我们来看一个更深入的例子。

协调与激励

假设我们要创建一个旅游网站，既需要获取游客目的地的列表，还需要获取地方事件的列表。这两个请求就是相互独立的 api 调用。

```
// Http.get :: String -> Task Error HTML

var renderPage = curry(function(destinations, events) { /* render page */  });

Task.of(renderPage).ap(Http.get('/destinations')).ap(Http.get('/events'))
// Task("<div>some page with dest and events</div>")
```

两个请求将会同时立即执行，当两者的响应都返回之后，`renderPage` 就会被调用。这与 monad 版本的那种必须等待前一个任务完成才能继续执行后面的操作完全不同。本来我们就无需根据目的地来获取事件，因此也就不需要依赖顺序执行。

再次强调，因为我们是使用局部调用的函数来达成上述结果的，所以必须要保证 `renderpage` 是 curry 函数，否则它就不会一直等到两个 `Task` 都完成。而且如果你碰巧自己做过类似的事，那你一定会感激 `applicative functor` 这个异常

简洁的接口的。这就是那种能够让我们离“奇点”（singularity）更近一步的优美代码。

再来看另外一个例子。

```
// 帮助函数：
// ==============
//  $ :: String -> IO DOM
var $ = function(selector) {
  return new IO(function(){ return document.querySelector(select
or) });
}

//  getVal :: String -> IO String
var getVal = compose(map(_.prop('value')), $);

// Example:
// ===============
//  signIn :: String -> String -> Bool -> User
var signIn = curry(function(username, password, remember_me){ /*
 signing in */  })

IO.of(signIn).ap(getVal('#email')).ap(getVal('#password')).ap(IO
.of(false));
// IO({id: 3, email: "gg@allin.com"})
```

`signIn` 是一个接收 3 个参数的 curry 函数，因此我们需要调用 `ap` 3 次。在每一次的 `ap` 调用中，`signIn` 就收到一个参数然后运行，直到所有的参数都传进来，它也就执行完毕了。我们可以继续扩展这种模式，处理任意多的参数。另外，左边两个参数在使用 `getVal` 调用后自然而然地成为了一个 `IO`，但是最右边的那个却需要手动 `lift`，然后变成一个 `IO`，这是因为 `ap` 需要调用者及其参数都属于同一类型。

lift

（译者注：此处原标题是“Bro, do you even lift?”，是一流行语，发源于健身圈，指质疑别人的健身方式和效果并显示优越感，后扩散至其他领域。再注：作者书中用了不少此类俚语或俗语，有时并非在使用俚语的本意，就像这句，完全就是为了好

玩。另，关于 lift 的概念可参看第 8 章。）

我们来试试以一种 pointfree 的方式调用 applicative functor。因为 `map` 等价于 `of/ap`，那么我们就可以定义无数个能够 `ap` 通用函数。

```
var liftA2 = curry(function(f, functor1, functor2) {
  return functor1.map(f).ap(functor2);
});

var liftA3 = curry(function(f, functor1, functor2, functor3) {
  return functor1.map(f).ap(functor2).ap(functor3);
});

//liftA4, etc
```

`liftA2` 是个奇怪的名字，听起来像是破败工厂里挑剔的货运电梯，或者伪豪华汽车公司的个性车牌。不过你要是真正理解了，那么它的含义也就不证自明了：让那些小代码块发生 lift，成为 applicative functor 中的一员。

刚开始我也觉得这种 2-3-4 的写法没什么意义，看起来又丑又没有必要，毕竟我们可以在 JavaScript 中检查函数的参数数量然后再动态地构造这样的函数。不过，局部调用（partially apply） `liftA(N)` 本身，有时也能发挥它的用处，这样的话，参数数量就固定了。

来看看实际用例：

```
// checkEmail :: User -> Either String Email
// checkName :: User -> Either String String

//  createUser :: Email -> String -> IO User
var createUser = curry(function(email, name) { /* creating... */
 });

Either.of(createUser).ap(checkEmail(user)).ap(checkName(user));
// Left("invalid email")

liftA2(createUser, checkEmail(user), checkName(user));
// Left("invalid email")
```

`createUser` 接收两个参数，因此我们使用的是 `liftA2` 。上述两个语句是等价的，但是使用了 `liftA2` 的版本没有提到 `Either` ，这就使得它更加通用灵活，因为不必与特定的数据类型耦合在一起。

我们试试以这种方式重写前一个例子：

```
liftA2(add, Maybe.of(2), Maybe.of(3));
// Maybe(5)

liftA2(renderPage, Http.get('/destinations'), Http.get('/events'
))
// Task("<div>some page with dest and events</div>")

liftA3(signIn, getVal('#email'), getVal('#password'), IO.of(false
));
// IO({id: 3, email: "gg@allin.com"})
```

操作符

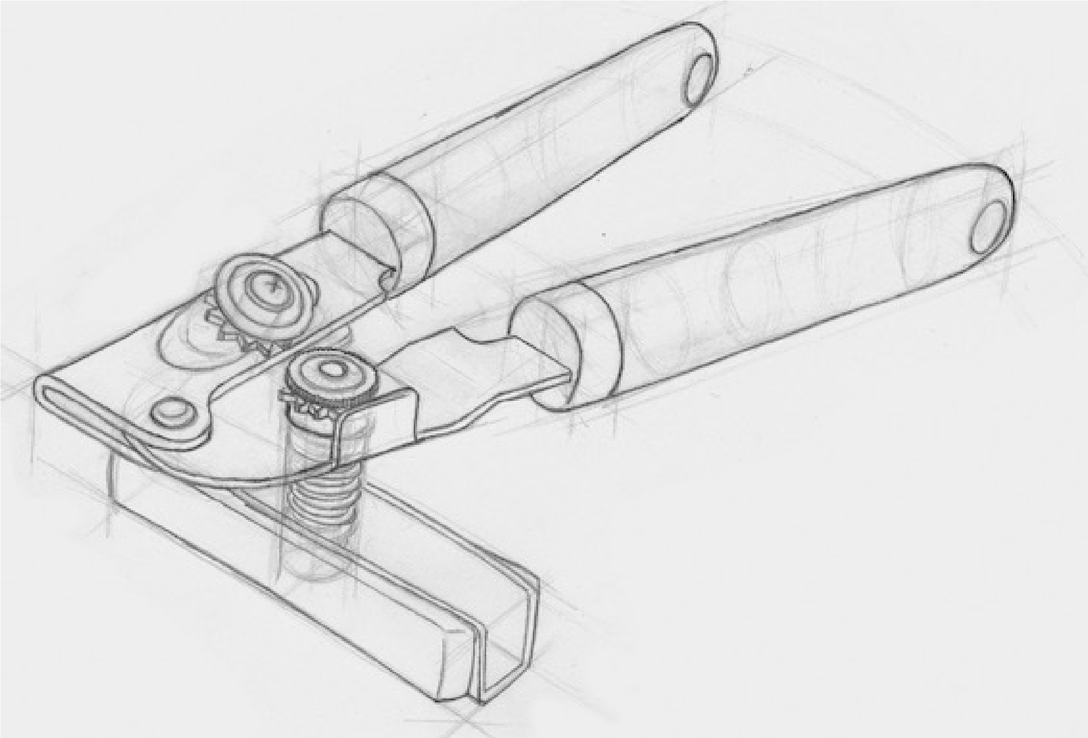
在 haskell、scala、PureScript 以及 swift 等语言中，开发者可以创建自定义的中缀操作符（infix operators），所以你能看到到这样的语法：

```
-- haskell
add <$> Right 2 <*> Right 3
```

```
// JavaScript
map(add, Right(2)).ap(Right(3))
```

`<$>` 就是 `map` （亦即 `fmap` ）， `<*>` 不过就是 `ap` 。这样的语法使得开发者可以以一种更自然的风格来书写函数式应用，而且也能减少一些括号。

免费开瓶器



我们尚未对衍生函数（derived function）着墨过多。不过看到本书介绍的所有这些接口都互相依赖并遵守一些定律，那么我们就可以根据一些强接口来定义一些弱接口了。

比如，我们知道一个 applicative 首先是一个 functor，所以如果已经有一个 applicative 实例的话，毫无疑问可以依此定义一个 functor。

这种完美的计算上的大和谐（computational harmony）之所以存在，是因为我们在跟一个数学“框架”打交道。哪怕是莫扎特在小时候就下载了 ableton（译者注：一款专业的音乐制作软件），他的钢琴也不可能弹得更好。

前面提到过， `of/ap` 等价于 `map` ，那么我们就可以利用这点来定义 `map` ：

```
// 从 of/ap 衍生出的 map
X.prototype.map = function(f) {
  return this.constructor.of(f).ap(this);
}
```

monad 可以说是处在食物链的顶端，因此如果已经有了一个 `chain` 函数，那么就可以免费得到 functor 和 applicative：

```
// 从 chain 衍生出的 map
X.prototype.map = function(f) {
  var m = this;
  return m.chain(function(a) {
    return m.constructor.of(f(a));
  });
}

// 从 chain/map 衍生出的 ap
X.prototype.ap = function(other) {
  return this.chain(function(f) {
    return other.map(f);
  });
};
```

定义一个 monad，就既能得到 applicative 也能得到 functor。这一点非常强大，相当于这些“开瓶器”全都是免费的！我们甚至可以审查一个数据类型，然后自动化这个过程。

应该要指出来的一点是， `ap` 的魅力有一部分就来自于并行的能力，所以通过 `chain` 来定义它就失去了这种优化。即便如此，开发者在设计出最佳实现的过程中就能有一个立即可用的接口，也是很好的。

为啥不直接使用 monad？因为最好用合适的力量来解决合适的问题，一分不多，一分不少。这样就能通过排除可能的功能性来做到最小化认知负荷。因为这个原因，相比 monad，我们更倾向于使用 applicative。

向下的嵌套结构使得 monad 拥有串行计算、变量赋值和暂缓后续执行等独特的能力。不过见识到 applicative 的实际用例之后，你就不必再考虑上面这些问题了。

下面，来看看理论知识。

定律

就像我们探索过的其他数学结构一样，我们在日常编码中也依赖 applicative functor 一些有用的特性。首先，你应该知道 applicative functor 是“组合关闭”（closed under composition）的，意味着 `ap` 永远不会改变容器类型（另一个胜过 monad

的原因）。这并不是说我们无法拥有多种不同的作用——我们还是可以把不同的类型压栈的，只不过我们知道它们将会在整个应用的过程中保持不变。

下面的例子可以说明这一点：

```
  var tOfM = compose(Task.of, Maybe.of);

  liftA2(_.concat, tOfM('Rainy Days and Mondays'), tOfM(' always get me down'));
  // Task(Maybe(Rainy Days and Mondays always get me down))
```

你看，不必担心不同的类型会混合在一起。

该去看看我们最喜欢的范畴学定律了：同一律（identity）。

同一律（identity）

```
// 同一律
A.of(id).ap(v) == v
```

是的，对一个 functor 应用 `id` 函数不会改变 `v` 里的值。比如：

```
var v = Identity.of("Pillow Pets");
Identity.of(id).ap(v) == v
```

`Identity.of(id)` 的“无用性”让我不禁莞尔。这里有意思的一点是，就像我们之前证明了的，`of/ap` 等价于 `map`，因此这个同一律遵循的是 functor 的同一律：`map(id) == id`。

使用这些定律的优美之处在于，就像一个富有激情的幼儿园健身教练让所有的小朋友都能愉快地一块玩耍一样，它们能够强迫所有的接口都能完美结合。

同态（homomorphism）

```
// 同态
A.of(f).ap(A.of(x)) == A.of(f(x))
```

同态就是一个能够保持结构的映射（structure preserving map）。实际上，functor 就是一个在不同范畴间的同态，因为 functor 在经过映射之后保持了原始范畴的结构。

事实上，我们不过是把普通的函数和值放进了一个容器，然后在里面进行各种计算。所以，不管是把所有的计算都放在容器里（等式左边），还是先在外面进行计算然后再放到容器里（等式右边），其结果都是一样的。

一个简单例子：

```
Either.of(_.toUpper).ap(Either.of("oreos")) == Either.of(_.toUpp
er("oreos"))
```

互换（interchange）

互换（interchange）表明的是选择让函数在 `ap` 的左边还是右边发生 lift 是无关紧要的。

```
// 互换
v.ap(A.of(x)) == A.of(function(f) { return f(x) }).ap(v)
```

这里有个例子：

```
var v = Task.of(_.reverse);
var x = 'Sparklehorse';

v.ap(Task.of(x)) == Task.of(function(f) { return f(x) }).ap(v)
```

组合（composition）

最后是组合。组合不过是在检查标准的函数组合是否适用于容器内部的函数调用。

```
// 组合
A.of(compose).ap(u).ap(v).ap(w) == u.ap(v.ap(w));
```

```
var u = IO.of(_.toUpper);
var v = IO.of(_.concat("& beyond"));
var w = IO.of("blood bath ");

IO.of(_.compose).ap(u).ap(v).ap(w) == u.ap(v.ap(w))
```

总结

处理多个 functor 作为参数的情况，是 applicative functor 一个非常好的应用场景。借助 applicative functor，我们能够在 functor 的世界里调用函数。尽管已经可以通过 monad 达到这个目的，但在不需要 monad 的特定功能的时候，我们还是更倾向于使用 applicative functor。

至此我们已经基本介绍完容器的 api 了，我们学会了如何对函数调用 `map`、`chain` 和 `ap`。下一章，我们将学习如何更好地处理多个 functor，以及如何以一种原则性的方式拆解它们。

Chapter 11: Traversable/Foldable Functors

练习

```
require('./support');
var Task = require('data.task');
var _ = require('ramda');

// 模拟浏览器的 localStorage 对象
var localStorage = {};



// 练习 1
// ==========
// 写一个函数，使用 Maybe 和 ap() 实现让两个可能是 null 的数值相加。

//  ex1 :: Number -> Number -> Maybe Number
var ex1 = function(x, y) {
```

```
};


// 练习 2
// ==========
// 写一个函数，接收两个 Maybe 为参数，让它们相加。使用 liftA2 代替 ap()。


//  ex2 :: Maybe Number -> Maybe Number -> Maybe Number
var ex2 = undefined;




// 练习 3
// ==========
// 运行 getPost(n) 和 getComments(n)，两者都运行完毕后执行渲染页面的操作。（参数 n 可以是任意值）。

var makeComments = _.reduce(function(acc, c){ return acc+"<li>"+c+"</li>" }, "");
var render = _.curry(function(p, cs) { return "<div>"+p.title+"</div>"+makeComments(cs); });

//  ex3 :: Task Error HTML
var ex3 = undefined;




// 练习 4
// ==========
// 写一个 IO，从缓存中读取 player1 和 player2，然后开始游戏。

localStorage.player1 = "toby";
localStorage.player2 = "sally";

var getCache = function(x) {
  return new IO(function() { return localStorage[x]; });
}
var game = _.curry(function(p1, p2) { return p1 + ' vs ' + p2; });
```

```
//  ex4 :: IO String
var ex4 = undefined;





// 帮助函数
// =====================

function getPost(i) {
  return new Task(function (rej, res) {
    setTimeout(function () { res({ id: i, title: 'Love them futu
res' }); }, 300);
  });
}

function getComments(i) {
  return new Task(function (rej, res) {
    setTimeout(function () {
      res(["This book should be illegal", "Monads are like space
 burritos"]);
    }, 300);
  });
}
```